

LEVEL

AGARD-CP-303

DTIC FILE COPY AD A109274
AGARD-CP-303

AGARD

ADVISORY GROUP FOR AEROSPACE RESEARCH & DEVELOPMENT

7 RUE ANCELLE 92200 NEUILLY SUR SEINE FRANCE

AGARD CONFERENCE PROCEEDINGS No. 303

Tactical Airborne Distributed Computing and Networks

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

DTIC
JAN 6 1982
H

NORTH ATLANTIC TREATY ORGANIZATION



DISTRIBUTION AND AVAILABILITY
ON BACK COVER

82 01 05 008

P

AGARD-CP-303

NORTH ATLANTIC TREATY ORGANIZATION
ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT
(ORGANISATION DU TRAITE DE L'ATLANTIQUE NORD)

AGARD Conference Proceedings No.303

**TACTICAL AIRBORNE DISTRIBUTED
COMPUTING AND NETWORKS**

71-
SELECTED
H

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution unlimited

Copies of papers and discussions presented at a Meeting of the Avionics Panel
held in Røros, Norway 22-25 June, 1981.

THE MISSION OF AGARD

The mission of AGARD is to bring together the leading personalities of the NATO nations in the fields of science and technology relating to aerospace for the following purposes:

- Exchanging of scientific and technical information;
- Continuously stimulating advances in the aerospace sciences relevant to strengthening the common defence posture;
- Improving the co-operation among member nations in aerospace research and development;
- Providing scientific and technical advice and assistance to the North Atlantic Military Committee in the field of aerospace research and development;
- Rendering scientific and technical assistance, as requested, to other NATO bodies and to member nations in connection with research and development problems in the aerospace field;
- Providing assistance to member nations for the purpose of increasing their scientific and technical potential;
- Recommending effective ways for the member nations to use their research and development capabilities for the common benefit of the NATO community.

The highest authority within AGARD is the National Delegates Board consisting of officially appointed senior representatives from each member nation. The mission of AGARD is carried out through the Panels which are composed of experts appointed by the National Delegates, the Consultant and Exchange Programme and the Aerospace Applications Studies Programme. The results of AGARD work are reported to the member nations and the NATO Authorities through the AGARD series of publications of which this is one.

Participation in AGARD activities is by invitation only and is normally limited to citizens of the NATO nations.

The content of this publication has been reproduced directly from material supplied by AGARD or the authors.

Published October 1981

Copyright © AGARD 1981
All Rights Reserved

ISBN 92-835-0302-3



*Printed by Technical Editing and Reproduction Ltd
Harford House, 7-9 Charlotte St, London, W1P 1HD*

THEME

A distributed processing system has been characterized as having a multiplicity of physically distributed resources interacting through a communication network; high-level operating system software unifies, controls, and integrates the components and provides transparency to services rendered.

The distributed system architecture offers cooperative autonomy in overall operation to achieve efficient use of avionic resources and to provide high system integrity, cost-effective maintenance, expandability, and improved performance. The physical distribution of resources comprising a system works to insure immunity to battle damage and accidents. Also, in some instances systems for distributed computation may take the form of air-to-satellite, air-to-surface, or air-to-air. The advent of small, inexpensive, low-power computing revolutionized complex systems design, and raises serious questions regarding the future of centralized, hardwired avionics computer systems.

Distributed processing, having been made possible by the price performance revolution in micro-electronics, now challenges us to correctly apply the concept to alleviate cost, schedule, reliability, operational, and maintenance problems in avionic systems.

Accession For		<input checked="checked" type="checkbox"/>
NTIS	CONF	<input type="checkbox"/>
DTIC	CONF	<input type="checkbox"/>
USNS	CONF	<input type="checkbox"/>
Distribution		
By		
Date		
Approved		
Special		
Dist		
A		

PROGRAM AND MEETING OFFICIALS

Chairman: Mr B.L.Dove, US

Program Committee: Mr W.F.Ball, US
Mr T.J.Sueta, US
Mr O.Rossignol, FR
Ir. H.A.Timmers, NE
Dr G. Van Keuk, GE
Mr R.Vaughn, US
Mr R.Wright, UK

LOCAL COORDINATOR

Dr L.Hoivik, NO
NDRE, Div. for Electronics
Kjeller, NO

AVIONICS PANEL

Chairman: Dr M.Vogel	Deputy Chairman: Mr Y.Brault
DFVLR e.v.	Thomson CSF
8031 Oberpfaffenhofen	Division Equipements
Post Wessling/obb	Avioniques et Spatiaux
FRG	178 Bld Gabriel Péri
	92240 Malakoff
	FR

PANEL EXECUTIVE

Lt.Col. J.B.Catiller
AGARD/NATO
7, rue Ancelle
92200 Neuilly-sur-Seine
France

CONTENTS

	Page
THEME	iii
PROGRAM AND MEETING OFFICIALS	iv
TECHNICAL EVALUATION REPORT by B.L.Dove	viii
	Reference

SESSION I - STATE-OF-THE-ART IN DISTRIBUTED PROCESSING

DISTRIBUTED DATA PROCESSING - WHAT IS IT? By P.H.Enslow, Jr.	1
Paper 2 cancelled	
THE EFFECT OF INCREASINGLY MORE COMPLEX AIRCRAFT AND AVIONICS ON THE METHOD OF SYSTEM DESIGN by J.T.Martin	3
A TUTORIAL ON DISTRIBUTED PROCESSING IN AIRCRAFT/AVIONICS APPLICATIONS by B.A.Zempolich	4
SUMMARY AND DISCUSSION	S1

SESSION II - DISTRIBUTED AIRBORNE SYSTEM ARCHITECTURE

Paper 5 cancelled	
PERFORMANCE STUDY OF A DISTRIBUTED MICROPROCESSOR ARCHITECTURE FOR USE ABOARD MILITARY AIRCRAFT by K.G.Shin and C.M.Krishna	6
THE DEVELOPMENT OF ASYNCHRONOUS MULTIPROCESSOR CONCEPTS FOR FLIGHT CONTROL SYSTEM APPLICATIONS by S.M.Wright and J.G.Brown	7
FUNCTIONAL VERSUS COMMUNICATION STRUCTURES IN MODERN AVIONIC SYSTEMS by K.Brammer and A.Weimann	8
CONTINUOUS RECONFIGURATION IN A MULTI-MICROPROCESSOR FLIGHT CONTROL SYSTEM by S.L.Maher and S.J.Larimer	9
EXPERIENCES WITH THE EXPERIMENTAL FFM-MCS by H.V.Issendorff	10
SUMMARY AND DISCUSSION	S2

SESSION III - DISTRIBUTED SYSTEM DESIGN APPROACHES

SAVANT - A DATABASE MANIPULATION TECHNIQUE FOR SYSTEM ARCHITECTURE DESIGN VERIFICATION ANALYSIS by A.A.Callaway	11
SIGNAL PROCESSING WITH SYSTOLIC ARRAYS by R.W.Priester, K.Bromley, J.Clary and H.Whitehouse	12

ECONOMIC CONSIDERATIONS FOR REAL-TIME NAVAL AIRCRAFT/AVIONIC DISTRIBUTED COMPUTER CONTROL SYSTEMS by B.A.Zempolich	13
--	----

FUNCTIONAL DOCUMENTATION - A PRACTICAL AID TO THE ORDERLY SOLUTION OF THE SYSTEM DESIGN PROBLEM by J.T.Martin	14
---	----

SUMMARY AND DISCUSSION	S3
------------------------	----

SESSION IV - DISTRIBUTED SYSTEM SOFTWARE

A CONSISTENT APPROACH TO THE DEVELOPMENT OF SYSTEM REQUIREMENTS AND SOFTWARE DESIGN by A.O.Ward	15
---	----

A PEARL SOFTWARE SYSTEM FOR MULTI-PROCESSOR SYSTEMS by P.Elzer and H.J.Schneider. Presented by Mr Schloch	16
--	----

DISTRIBUTED AND DECENTRALIZED CONTROL IN FULLY DISTRIBUTED PROCESSING SYSTEMS by P.H.Enslow, Jr. Presented by Dr Livesey	17
--	----

RECOVERY IN DISTRIBUTED PROCESSING SYSTEMS by L.Svobodova	18
--	----

GENERALIZED POLLING ALGORITHMS FOR DISTRIBUTED SYSTEMS by J.K.Wolf	19
---	----

SUMMARY AND DISCUSSION	S4
------------------------	----

SESSION V - FAULT TOLERANCE AND RELIABILITY IN DESIGNS

STAGE-STATE RELIABILITY ANALYSIS TECHNIQUE by A.D.Stem	20
---	----

METHODOLOGY FOR MEASUREMENT OF FAULT LATENCY IN A DIGITAL AVIONIC MINIPROCESSOR by J.G.McGough, F.Swern and S.J.Bavuso. Presented by Mr Moses	21
---	----

HIERARCHICAL SPECIFICATION OF THE SIFT FAULT TOLERANT FLIGHT CONTROL SYSTEM by P.M.Melliar-Smith and R.L.Schwartz	22
---	----

RECONFIGURATION: A METHOD TO IMPROVE SYSTEMS RELIABILITY by J.Szlachta	23
---	----

RESEAU D'ECHANGE RECONFIGURABLE POUR CONTROLE DE PROCESSUS REPARTI par Ch.Meraud et B.Maurel	24
---	----

SUMMARY AND DISCUSSION	S5
------------------------	----

SESSION VI - INTERCONNECTION - BUSSING AND NETWORKING

Paper 25 cancelled

PROTOCOL LEVEL MODULES - FOR COST EFFECTIVE STANDARD COMPUTER COMMUNICATION by Q.Hvinden, Y.Lundh and Q.Sandholt	26
--	----

	Reference
LES STRATEGIES DE RETRANSMISSION POUR LE CONTROLE D'ERREUR DANS LES PROTOCOLES DE TRANSFERT DE DONNEES par G.Juanole	27
PRACTICAL ASPECTS WHICH APPLY TO MIL-STD-1553B DATA NETWORKS by I.Moir and P.A.Duke	28
THE TRAFFIC FLOW IN A DISTRIBUTED REALTIME COMPUTING SYSTEM (RDC-SYSTEM) WITH A FIBER OPTIC RINGBUS SYSTEM by D.Heger and R.Bähre	29
DISPERSED SENSOR PROCESSING MESH PROJECT by V.A.Megna	30
NEXT GENERATION MILITARY AIRCRAFT WILL REQUIRE HIERARCHICAL/MULTI- LEVEL INFORMATION TRANSFER SYSTEMS by J.W.McCuen	31
SUMMARY AND DISCUSSION	S6
 <u>SESSION VII - APPLICATION OF DISTRIBUTED SYSTEM DESIGNS TO AVIONIC SYSTEMS</u> 	
SiFT - AN ULTRA-RELIABLE AVIONIC COMPUTING SYSTEM by K.Moses	32
STATE-OF-THE-ART COMPUTER MONITORING EQUIPMENT by H.Nelson	33
INTEGRATED CONTROL OF MECHANICAL SYSTEMS FOR FUTURE COMBAT AIRCRAFT by G.W.Wilcock, P.Lancaster and C.Moxey	34
ARCHITECTURE DU SYSTEME D'ARMES DU MIRAGE 2000 par S.Croce-Spinelli, B.Vandecasteele et J.F.Ferri	35
THE COMPUTER SYSTEM OF THE TORNADO* by P.A.Bross	36
F/A-18A TACTICAL AIRBORNE COMPUTATIONAL SUBSYSTEM by T.V.McTigue	37
F/A-18 WEAPONS SYSTEM SUPPORT FACILITIES by T.F.C'Neill	38
SUMMARY AND DISCUSSION	S7
LIST OF ATTENDEES	A

*Paper RESTRICTED. Copies available from author, see List of Attendees.

TACTICAL AIRBORNE DISTRIBUTED
COMPUTING AND NETWORKS

TECHNICAL EVALUATION REPORT

Billy L. Dove
Technical Program Chairman

EXECUTIVE SUMMARY -

CONCLUSIONS

- o Benefits credited to distributed data processing are not capable of being realized within the current state-of-the-art.
- o Preparation of military standards for airborne distributed data processing is inappropriate at this time. However, a mechanism to promote uniformity in technical definitions between workers in this area would be useful.
- o The state-of-the-art is not adequate to support the design and validation of airborne distributed data processing systems for critical military missions.
- o The economic leverage of the military is no longer a factor with microelectronic manufacturers, therefore, system designers must consider technology independence in their designs.
- o Software is of considerable importance to this area.

RECOMMENDATIONS

- o AGARD follow up on this subject area with a future meeting.
- o AGARD support specialist meeting on Methodology and Design Techniques for Distributed Systems.

GENERAL -

The symposium was three and a half days in length being held June 22 to 25, 1981, in Roros, Norway.

Approximately 130 people were registered. Attendance at all sessions was unusually high.

Thirty-eight papers were scheduled for presentation as of June 22, 1981, and only two papers were not presented at the meeting.

Few of the papers were invited ones. Even so, the material gathered for the program proved to be of interest overall.

A large number of questions were asked whose answers are contained in the proceedings.

A major objective of this meeting was to seek a delineation of the state-of-the-art in airborne distributed computing. Fortunately, this symposium attracted a large number of people representing a broad range of interests and included academics, institutes, and avionic and airframe manufacturers. This was as intended by the program committee.

TECHNICAL SESSIONS -

The potential benefits from distributed computing system concepts such as improved reliability/availability, ease of system growth, shared resources, etc., offer attractive alternatives to today's problems, however, the technical capability to realize these benefits has been brought into question. Thus, the purpose of the meeting was established—to assess the state-of-the-art capability in airborne distributed data processing.

The meeting was organized in such a way as to encourage a diverse response from the call for papers. Seven sessions were defined, as follows:

Session I	State-of-the-Art
Session II	Architectures
Session III	Design Approaches
Session IV	Software
Session V	Fault Tolerance and Reliability
Session VI	Bussing and Networking
Session VII	Applications

Session I: State-of-the-Art in Distributed Processing. This was a tutorial session. The first paper was invited and given extra time. It focused on the definition of distributed computing systems. The matter of definition is important as it relates a name to a level of potential benefits. This proved to be a very interesting and much needed paper as judged by the reaction of the audience. A continuing need for the refinement of technical definitions was established. A major point from this session was that the state-of-the-art is far from being able to provide the benefits claimed by airborne distributing systems enthusiasts.

The introductory paper (1) written by Dr. Philip Enslow, USA, "Distributed Data Processing--What Is It?" was presented by Dr. John Livesey. The paper focused on definitions which set the scene for the entire symposium.

Mr. Martin's paper (3), "The Effect of Increasingly More Complex Aircraft and Avionics on the Method of System Design," presented a historical treatment of aircraft and their systems. His point being that little change was required in the design methodology for systems of the past, but that a revolutionary change in methods is required in order to design distributed systems.

Mr. Zempolich's paper (4), "A Tutorial on Distributed Processing in Aircraft/Avionics Applications," dealt with systems architectural concepts from the analog to digital and beyond to the hierarchical. Emphasis was placed upon the need for top-down design and the synergism achievable from a team approach.

Following the three tutorial papers of the first session, the authors and Dr. Von Issendorff participated in a free-exchange question and answer period. Arising from the many questions and comments during this period was the subject of military standards and academic definitions.

Session II: Distributed Airborne System Architecture. The papers of this session revealed a tendency to exploit the advances made in microcomputer technology by partitioning both hardware and software into functional modules. The drivers for this are: a possible positive influence on reliability and damage tolerance; use of identical hardware; cost of software; and better visibility into the system for better maintainability. It was abundantly clear that system architects are at work putting new technology to use in new ways. It is also clear that the resulting architectures are in general following the same trend, i.e., distributed microprocessors, bus connected, and with some form of dynamic redistribution of resources or functions. It seems logical and can be so argued that these architectures offer benefits in cost and reliability. It was not established from this session that a body of data exist which quantifies design factors and substantiates the claims made for the various category of architectures presented.

The paper by Dr. Shin (6), "Performance Study of a Distributed Microprocessor Architecture for Use Aboard Military Aircraft," proposed a concept based upon the decomposition of a mission into "atom functions" to be implemented by microelectronic technology. A central controller communicates with the "atom functions," and the pilot interfaces with the central controller. A hypothetical system was studied and some performance data generated.

Mr. Wright's paper (7), "The Development of Asynchronous Multiprocessor Concepts for Flight Control System Applications," describes a concept for the use of multiple microprocessors, functionally dedicated, and running asynchronously. The concept will be implemented and flown on a Hunter aircraft as a fly-by-wire system.

Mr. Brammer's paper (8), "Functional Versus Communication Structures in Modern Avionic Systems," presented results from investigations into the amount of interconnections in several avionic system concepts. The bus concept and the layered star appear to offer less interconnections from this analysis.

Lt. Maher's paper (9), "Continuous Reconfiguration in a Multi-Microprocessor Flight Control System," offered another concept of microcomputers interconnected with busses and an algorithm to dynamically redistribute system functions. Three advantages of this architecture were offered: expandability, reduction of software cost, and reduction of unscheduled maintenance.

Dr. Von Issendorff's paper (10), "Experiences with the FFM-MCS," presented the design of a test-bed for research on distributed data processing. Research tasks undertaken include decomposition of data processing tasks into sets of functions, message construction, and transfer protocol.

Session III: Distributed System Design Approaches.

Dr. Callaway's paper (11), "SAVANT - A Database Manipulation Technique for System Architecture Design Verification and Analysis," described an interactive tool capable of representing the various facets of a digital system design. SAVANT traces errors, identifies inconsistencies in designs, and provides data for trade-off between different configurations.

Dr. Whitehouse's paper (12), "Signal Processing with Systolic Arrays," presented a specialized hardware approach for fast matrix computation.

Mr. Zempolich's paper (13), "Economic Considerations for Real-Time Naval Aircraft/Avionic Distributed Computer Control Systems," emphasized the economic aspects to be considered during system design. Lack of economic leverage over microelectronic manufacturers has resulted in questions about the viability of standardization and commonality.

Mr. Martin's paper (14), "Functional Documentation - A Practical Aid to the Orderly Solution of the System Design Problem," discussed an organized approach to the decomposition of system requirements from specification to functional detail. This technique promotes communication between persons of different disciplines, and results in a well-documented design.

Session IV: Distributed System Software. Considering the criticality of software to the realization and success of distributed systems, it was surprising that this session received the least support in papers. The individual papers were of sufficient quality; however, the scope and number were inadequate. A final conclusion cannot be drawn regarding the state-of-the-art of software for distributed systems.

Mr. Ward's paper (15), "A Consistent Approach to the Development of System Requirements and Software Design," reported on the SAFRA (Semi-Automatic Functional Requirements Analysis) project. Many of the ingredients of a careful analysis of requirements and their mechanization were discussed. No comparison of SAFRA to other approaches was mentioned. A limited experience base exists with SAFRA.

Dr. Livesey's paper (17), "Distributed and Decentralized Control in Fully Distributed Systems," reinforced the definition of fully distributed systems through the discussion of decentralized control. An important aspect of the paper was the task graph concept. Information stored in task graphs could be useful to implementing the dynamics of reconfiguration.

Dr. Svobodova's paper (18), "Recovery in Distributed Processing," discussed techniques for analyzing task handling, confinement of failure effects, preservation of status, and recovery in digital systems.

Dr. Wolf's paper (19), "Generalized Polling Algorithms for Distributed Systems," compared two polling algorithms with the objective of eliminating the inefficiency of round-robin polling. This theoretical paper offered no example to illustrate the amount of improvement made.

Session V: Fault Tolerance and Reliability in Designs. Three papers in this session demonstrated great breadth in the consideration being given reliability assessment and validation. Emphasis on ultrareliability and the validation problems for such systems is being worked in the civil R&D sector.

Mr. Stearn's paper (20), "State-State Reliability Analysis Technique," presented an improved method for reliability analysis for redundant systems. The state-state technique is less difficult to use and simpler, thus not as many errors will be caused by having such a large number of combinations in the analysis. Sources of unreliability become readily apparent using this technique.

Mr. Moses' paper (21), "Methodology for Measurement of Fault Latency in a Digital Avionic Multiprocessor," discussed the use of an emulator to conduct fault injection experiments. The results from this work brings into question the fault/failure detection capability of self-test programs in avionic systems, and the accuracy of reliability analysis program results.

Dr. Schwartz's paper (22), "Hierarchical Specification of SIFT Fault Tolerant Flight Control System," offered for consideration a formal mathematical proof of ultrareliable computer functional and reliability requirements. This approach establishing a mathematically provable relationship between the specification and the programs of the actual systems is an intriguing and novel approach.

Mr. Szlachta's paper (23), "Reconfiguration: A Method to Improve Systems Reliability," discussed the improvement of reliability by use of hardware reconfiguration.

Mr. Meraud's paper (24), "Reseau d'Echange Reconfigurable pour Controle de Processus Reparti," discussed a means for dynamic distributed (decentralized) control of reconfiguration. This technique is directed to systems of high reliability although no reliability analysis results were given.

Session VI: Interconnection - Bussing and Networking.

Mr. Hvinden's paper (26), "Protocol Level Modules—For Cost-Effective Standard Computer Communications," describes a "host independent" implementation of computer communication protocols. This is realized by the development of hardware and software modules. Workload on the host is reduced.

Mr. Juanole's paper (27), "Les Stratégies de Retransmission pour le Contrôle d'Erreur dans les Protocoles de Transfert de Données," presented the functions of a protocol for data transfer. Error control is included in an error detection and data retransmission scheme. This strategy was analyzed and the logic of the process explained.

Mr. Duke's paper (28), "Practical Aspects Which Apply to MIL-STD-1553B Data Networks," discussed the ramifications of trying to satisfy two different standards—one relating to data transmission and the other to standard interfacing electronics. This situation is created when bus redundancy, multibus architecture, and some intelligence is required in a stores management and weapons aiming system.

Mr. Heger's paper (29), "The Traffic Flow Measured in a Distributed Real Time Computing System (RDC) With a Fiber Optic Ring Bus System," presented results from analysis and measurements taken from a fault tolerant microprocessor system's fiber optic ring bus. This is a very good example from which to compare the results of an analytical method vs. practical data gathering. Many more examples will be required to validate either method.

Mr. Megna's paper (30), "Dispersed Sensor Processing Mesh Project," presented the mechanization of a limited port network communication structure. Although detailed in implementation, neither the general analytical methods nor the general study results were presented which compared performance to the existing F-8.

Mr. McCuen's paper (31), "Next General Military Aircraft Will Require Hierarchical/Multi-Level Information Transfer Systems," was concerned with a discussion of a future high-speed data bus standard and architectural approaches to it. Information was requested to assist the task group in the formulation of a high order transfer-type system.

Session VII: Application of Distributed System Designs to Avionics Systems.

Mr. Moses' paper (32), "SIFT - An Ultra-Reliable Avionic Computing System," noted that in recent years automatic flight control systems (FCS) in aircraft, which previously provided mainly stability augmentation and other pilot-relief functions, have more recently taken on flight-critical tasks. These flight-critical tasks are those whose successful accomplishment is vital to the safety of the aircraft (e.g., automatic landing, fly-by-wire control system, control-figured vehicle methods). An FCS which takes on these critical safety-related tasks must be ultrareliable. SIFT implements software-implemented fault-tolerance techniques utilizing hardware redundancy. Achievement of failure probabilities of 10^{10} per hour were quoted. Using multiprocessor "star connection" techniques, the computing is carried out by high-speed 10-bit Bendix 950 processors (each with a throughput of 800 KOPS with an appropriate FCS instruction mix and with 32K memory). Software algorithms are used for failure detection. After fault detection and isolation, the software provides reconfiguration to accommodate the fault. The paper described the SIFT architecture and its hardware implementation. As an efficient approach to the design of ultrareliable avionics, the author noted that it should pave the way for acceptance of fly-by-wire and other advanced FCS techniques.

Mr. Nelson's paper (33), "State-of-the-Art Computer Monitoring Equipment," described the results of a significant software support effort at the NAVWPNCEN that has resulted in the availability and practical use of an airborne-computer hardware monitor. This device, called SOVAC (Software Validation And Control) provides a high capacity, real-time and user-selective "window" that gives high visibility into the internal operation of the tactical computer. SOVAC is a computer monitor that can conceptionally be thought of in terms of its basic components. These are: (1) Tactical Computer Interface. This section provides real-time control of the tactical computer and provides the capability to capture information available on the tactical computer's bus and control lines. (2) SOVAC Controller. This high-speed, microprogrammed controller coordinates the operation of the various subsystems. It has the capability to recognize various types of events or complex combinations of events and set a breakpoint. It has a very flexible data selection and logging capability. The functions of the controller are under the control of the user. (3) User Interface. This part of the SOVAC is the part that the operator actually uses. Its primary components are: a minicomputer, a terminal, an interface to the SOVAC controller and the SOVAC software. The paper noted that SOVAC is a powerful tool for use by anyone who has a need to know what is happening inside a tactical computer.

Mr. Wilcock's paper (34), "Centralized Management of Mechanical Systems for Future Combat Aircraft," described a computer oriented approach to the management of aircraft mechanical systems (fuel management, engine control, etc.). The approach described was a microprocessor-based management system distributed throughout the airframe. It is planned that these Systems Management Processors will operate independently as separate computing centers and will be interconnected via a data bus (MIL-STD-1553B or a derivative). Some of these microprocessors will act as remote terminals forwarding raw data via the bus to designated processing points. The paper described the various mechanical systems to be controlled, detailed the system architecture, described the mechanical system interface with the microprocessor and speculated on the cockpit displays and pilot interface. The system approach was seen to not only utilize current technology, but can take advantage of future technology and can be adapted at a reasonable cost and schedule to meet changing system requirements.

Mr. Vandecastelle's paper (35), "Architecture Du System D'armes Mirage 2000," stated that the architecture of the armament system of the Mirage 2000 represents an advanced generation of digital systems. It was described from the points of view of digital equipment, assignment of software to the equipment, digital links, and monitoring the system in flight. The paper discussed architectural principles that embraced hardware, software, the distribution of tasks, and corresponding interfaces. It is flexible enough to allow for the development of a family of systems of different sizes and different operational needs. It was noted that the architecture can be grossly characterized by the use of digital multiplex links of the "Digibus" type, a standard for French military aeronautics since 1974. In outline, the paper gave a general view of the Mirage 2000 system, including (1) a discussion of the principal sensors (navigation, radar, EO, active and passive countermeasures); (2) display and controls, all linked together by the standard Digibus technique; (3) the philosophy for the distribution of the compute tasks; (4) discussion of integrated and centralized functions;

(5) architectures for the central computers and the Digibus; and (6) the development methodology for the software.

Mr. Bross' paper (36), "Computer System of the Tornado," described the Tornado, while not representative of a modern distributed computer system, nevertheless, as a system with physically and functionally distributed computing power operating through a dense digital network. The end result is therefore a highly integrated system. The paper described the computing system and its architecture, viewing especially the functional autonomy of various subsystems. The provision of system integrity and fault tolerance incorporating redundancy and monitoring capability was highlighted. The Tornado's computer system was seen to be of a hierarchical distributed system type instead of an equally distributed mechanization. The (inner) lower part, serves for aircraft stability purposes, comprises the least intelligence but is highly redundant; the middle part provides for basic autopilot functions, is less redundant; the upper part serves for mission functions/modes and is simplex only. However, these highest mission functions are divided into two master functions, one as master for the horizontal (steering) and one for the vertical plane (terrain following). The paper concluded with "lessons-learned" and remarked on improvements to be considered for a next generation avionic system.

Mr. McTigue's paper (37), "F/A-18 Tactical Airborne Computer and Subsystem," presented a description of the Tactical Airborne Computational Subsystem used in the U.S. Navy/McDonnell Douglas F/A-18A Hornet Fighter/Attack Weapons System. The F/A-18A Hornet tactical computer subsystem consists of two central mission computers and a number of distributed processors embedded in various sensors and display subsystems. This distributed processing system is interconnected by and communicates over a MIL-STD-1553A serial 1-MHz command/response multiplex network. The distributed processing system architecture was discussed and the rationale was presented for the partitioning of the computational tasks between the central mission computers and the distributed processors embedded in the sensor subsystems. The salient features of the central mission computer and the distributed processors were discussed along with a description of the functional operation of the interconnecting MIL-STD-1553A multiplex communications system. Finally, the development process for the Operational Flight Program (OFP) for the central mission computers was described, including a discussion of the support facilities, which were used for the software integration and validation.

Mr. O'Neill's paper (38), "F/A-18 Weapon System Support Facilities," described the support facility tools being developed by the Navy. The U.S. Navy is currently acceptance-testing the McDonnell Douglas F/A-18 aircraft, which is an all-weather, fighter/attack aircraft with more than 30 on-board computers containing more than 700K words of programs. Since the F/A-18 is so much more complex than any aircraft currently deployed, sophisticated tools will be required by the system engineers to support the avionics. According to his paper, the F/A-18 Weapons System Support Facility (WSSF) at the NAVWPNCEN, China Lake, CA will contain all of the support tools (both hardware and software) necessary to test, modify, generate, and validate all of the avionics software, hardware, and firmware. The WSSF uses several minicomputers tied together in a distributed network to provide a realistic simulation of the aircraft flight characteristics. Using this approach, the avionics computers can be integrated into the simulation and tested in the WSSF before flight testing starts. The WSSF appears to be well under way in development and should ease the Navy's task of supporting the F/A-18 aircraft.

DISTRIBUTED DATA PROCESSING --- WHAT IS IT?

PHILIP H. ENSLOW JR.
 GEORGIA INSTITUTE OF TECHNOLOGY
 SCHOOL OF INFORMATION AND COMPUTER SCIENCE
 ATLANTA, GEORGIA 30332

SUMMARY

Distributed processing has been presented as the means to obtain improvements in a number of areas of system performance. Utilizing a list of these desired improvements as the motivational factors, this paper presents the key design characteristics of systems that will deliver a major proportion of these improvements. Because of the wide use of the term "distributed processing," the systems described here are identified as "fully distributed."

1 BACKGROUND

1.1 Goals of Computer System Development

Although the state of the art in digital computers has certainly been advancing faster than any other technological area in history, it is somewhat remarkable that the goals motivating most computer system development projects have remained basically unchanged since the earliest days. Perhaps the most important of these long sought-after improvements are the following:

1. Increased system productivity
 - Greater capacity
 - Shorter response time
 - Increased throughput
2. Improved reliability and availability
3. Ease of system expansion and enhancement
4. Graceful growth and degradation
5. Improved ability to share system resources

The "final or ultimate values" for these various goals cannot be expressed in absolute numbers, so it is not surprising that they continue to apply even though phenomenal advances have been made in many of them such as speed, capacity, and reliability. What is perhaps more noteworthy and important to the discussion being presented here is how little progress has been made in areas such as easy modular growth, availability, adaptability, etc.

It seems that each new major systems concept or development (e.g., multiprogramming, multiprocessing, networking, etc.) has been presented as "the answer" to achieving all of the goals listed above plus many others. "Distributed processing" is no exception to this rule. In fact, many salesmen have dusted off their old lists of benefits and are marketing today's distributed systems as the means to achieve all of them. Table 1 lists some of the benefits currently being claimed for distributed processing systems in current sales literature. Although some forms of distributed processing appear to offer great promise as a possible means to make significant advances in many of the areas listed, the state-of-the-art, particularly in system control software, is far from being able to deliver even a significant proportion of these benefits today.

1.2 Approaches to Improving System Performance

Efforts to improve the performance of digital computer systems can address or be focused on a number of major levels or design issues within the overall computer structure. These levels are:

1. Materials - the basic materials used in the construction of operating devices such as transistors, integrated circuits, or other switching devices.
2. Devices - operating devices such as transistors, integrated circuits, junctions, etc.
3. Switching circuits - design of circuits that provide fast and reliable logic operations.
4. Register-transfer - assemblies such as registers, buses, shift registers, adders, etc.
5. System architecture - algorithms for executing the basic functions such as arithmetic and logic operations, interrupt mechanisms, control of processor and memory states, etc.
6. System organization - the interconnection of major functional units such as control, memory, I/O, arithmetic/logic units, etc., and the rules governing the flow of data and control signals between these units. This level also considers the implementation of multiple, parallel paths for simultaneous operations and transfers.
7. Network organization - the number, characteristics, and topology of the interconnection of "complete" systems and the rules governing the control and utilization of the resources those systems provide.
8. System software - control and support software for the effective management and utilization of the hardware capabilities provided.

From the very beginning of the computer era there has been activity at all of these levels and such work continues today. (To place it into proper perspective, it should be noted that the research work carried on under this project is focused primarily at the three highest levels, system organization, network organization, and system software, with some work at level 5, system architecture.)

1.3 Parallel Processing Systems

An important theme of computer system development work at levels 5-8, "system architecture," "system organization," "network organization," and "system software," has been parallel processing. Parallel processing has been implemented utilizing approaches focused primarily on the system hardware or the software as well as integrated systems design.

Since the early days of computing, a direction of research that has offered high promise and attracted much attention is "parallel computing." Work in this area dates from the late 1950's which saw the development of the PILOT system [Lein58] at the National Bureau of Standards. The PILOT system consisted of "three independently operating computers that could work in cooperation" [Ensl74]. (From the information available, it appears that PILOT would be classified as a "loosely-coupled system" today.) It is interesting to note that the evolution of parallel "hardware" systems lead primarily to the development of tightly-coupled systems such as the Burroughs B-825 and B-5000, the earliest examples of the classical multiprocessor. Other development paths saw the introduction of specialized hardware systems such as SOLOMON and the ILLIAC IV, examples of other forms of tightly-coupled processors.

1.3.1 System Coupling

System coupling refers to the means by which two or more computer systems exchange information. It refers to both the physical transfer of such data as well as the manner in which the recipient of the data responds to its contents. These two aspects of system interconnection are called "physical coupling" and "logical coupling," and they are present in all multiple component systems whether the components of interest are complete computers or some smaller assembly.

The terms, "tight" and "loose" have been utilized to describe the mode of operation of each type of coupling. (Some authors have utilized a third category "medium coupling" and related it to a range of data transfer speeds; however, history has clearly shown that basing any characterizations of digital computers on speed, size, or even cost is an incorrect approach.) The interconnection and interaction of two computer systems can then be described by specifying the nature of its physical coupling and the nature of its logical coupling. It is important to point out that all four combinations of these characteristics are possible and that they all have been observed in implemented systems.

1.3.1.1 Tightly-Coupled Computer Systems

During the 1960's and 1970's, activities in the development of parallel computing, specifically multiple computer systems, were focused primarily on the development of tightly-coupled systems. These tightly-coupled systems took the form of classical multiprocessors (i.e., shared main memory) as well as specialized computation systems such as vector and array processors. This tight physical coupling resulted in a sharing of the directly executable address space common to both processors. There was no means by which the recipient of the data or information being transferred could refuse to physically accept it --- it was already there in his address space.

These early systems also usually implemented tight logical coupling. In this form of system interaction, the recipient of a message is required to perform whatever service is specified therein. With tight logical coupling, there is no independence of decision allowed regarding the performance of the service or activity "requested." The relationship between the sender and recipient is basically that of master-slave.

Although the concept of tightly-coupled multiprocessor systems appears to be a viable approach for achieving almost unlimited improvements in performance (i.e., increases in system throughput) with the addition of more processors, such has not been the results obtained with implemented systems. It is the very nature of tight-coupling that results in limitations on the improvements achievable. Some of the ways that these limitations have manifested themselves are listed below.

1. The direct sharing of resources (memory and input/output primarily) often results in access conflicts and delays in obtaining use of the shared resource.
2. User programming languages that support the effective utilization of tightly-coupled systems have not been adequately developed. The programmer must still be directly involved in job and task partitioning and the assignment of resources.
3. The development of "optimal" schedules for the utilization of the processors is very difficult except in trivial or static situations. Also, the inability to maintain perfect synchronization between all processors often invalidates an "optimal" schedule soon after it has been prepared.
4. Any inefficiencies present in the operating system appear to be greatly exaggerated in a tightly-coupled system.

There was also significant activity during these earlier periods in the development of multiple computer systems characterized as "attached support processors (ASP)." These systems were physically loosely-coupled; but, logically, they were tightly-coupled. The earliest examples of this type of system organization were the use of attached processors dedicated to input/output operations in large-scale batch processing systems. In the latter part of the 1970's, specialized vector and array processors as well as other special-purpose units such as fast Fourier transform units were being connected to general computational systems and utilized as attached support processors. In any event, the specialized nature of the services provided by the attached processor excludes them from

consideration as possible approaches to providing general-purpose computational support such as that available from tightly-coupled general-purpose processors functioning as multiprocessors.

Tightly-coupled systems certainly do have a role to play in the total spectrum of computer systems organization; however, their limitations should certainly be considered. It was the recognition of these limitations and the small amount of progress made in overcoming them despite the expenditure of very large research efforts that contributed to the decision to focus our current research program on loosely-coupled systems.

1.3.1.2 Loosely-Coupled Systems

Loosely-coupled systems are multiple computer systems in which the individual processors both communicate physically and interact logically with one another at the "input/output level." There is no direct sharing of primary memory, although, there may be sharing of an on-line storage device such as a disk in the interconnecting input/output communication path. The important characteristic of this type of system organization and operation is that all data transfer operations between the two component systems are performed as input/output operations. The unit of data transferred is whatever is permissible on the particular input/output path being utilized; and, in order to complete a transfer, the active cooperation of both processors is required (i.e., one might execute a READ operation in order to accommodate or accept another's WRITE).

Probably the most important characteristic of loose logical coupling is that one processor does not have the capability or authority to "force" another processor to do something. One processor can "deliver" data to another; however, even if that data is a request (or a "demand") for a service to be performed, the receiving processor, theoretically, has the full and autonomous rights to refuse to execute that request. The reaction of processors to such requests for service is established by the operating system rules of the receiving processor, not by the transmitter. This allows the recipient of a request to take into consideration "local" conditions in making the decision as to what actions to take. It is important to note that it is possible for a system to be physically loosely-coupled but logically tightly-coupled due to the rules embodied in the component operating systems, e.g., a permanent master/slave relationship is defined. The other reverse condition, tight physical and loose logical coupling, is also possible.

1.3.2 Computer Networks

A computer network can be characterized as a physically loosely-coupled, multiple-computer system in which the interconnection paths have been extended by the inclusion of data communications links. Fundamentally there are no differences between the basic characteristics of computer network systems and other loosely-coupled systems other than the data transfer rates normally provided. The transfer of data between two nodes in the network still requires the active cooperation of both parties involved, but there is no inherently required cooperation between the operation of the processors other than that which they wish to provide.

1.3.3 Distributed Systems

Although there is a large amount of confusion, and often controversy, over exactly what is a "distributed system," it is generally accepted that a distributed system is a multiple computer network designed with some unity of purpose in mind. The processors, databases, terminals, operating systems, and other hardware and software components included in the system have been interconnected for the accomplishment of an identifiable, common goal. That goal may be the supplying of general-purpose computing support, a collection of integrated applications such as corporate management, or embedded computer support such as a real-time process control system.

This research program is concerned with a very specific subclass of all of the systems currently being designated "distributed." The environment of interest here has been given the title "Fully Distributed Processing System" or FDPS. Section 2 discusses the general characteristics of FDPS's.

2 INTRODUCTION TO FULLY DISTRIBUTED PROCESSING SYSTEMS

2.1 Motivation of the FDPS Concept

A large number of claims have been made as to the benefits that will be achieved with distributed processing systems. As pointed out above, this list is very similar to the lists of "benefits to be achieved" with several earlier computer technologies. However, each of those earlier solutions failed to deliver its promises for various reasons. It was an examination of the "weaknesses" in the earlier concepts and the development of a set of principles to overcome these obstacles that led to the concept of "Fully Distributed Processing Systems" or as it is commonly referred to "FDPS."

The principle of parallel (i.e., simultaneous and/or concurrent) operation of a multiplicity of resources continues to be perhaps the most important goal. The unique feature of FDPS's is the means or environment in which this is attempted. A distributed system should exhibit a continual increase in performance as additional processing components are added. The users should observe shorter response times as well as an increase in total system throughput. In addition, the utilization of system resources should be higher as a result of the system's ability to perform automatic load balancing, servicing a large quantity and variety of user work requests. A distributed system should also permit the sharing of data between cooperating users and the making available of specialized resources found only on certain processors. In general, a distributed system should provide more facilities and a wider variety of services than those that can be offered by any system composed of a single processor [Hopp79]. Another important and highly desirable feature of such a system is extensibility. Extensibility might be realized in several different ways. The system might support modular and incremental growth permitting flexibility in its configuration, or it might support expansion in capacity, adding new functions, or both. Finally, it might provide for incremental replacement and/or upgrading of system components, either hardware or software. The executive control of the system is obviously the key to attaining these goals, and it is in the area of executive control that some of the most significant deficiencies of earlier systems have been found.

The major weaknesses in the executive control of earlier forms of parallel systems appear to result from an excessive degree of centralization of control functions reflected in centralized decision making or centralized maintenance of system status information or both of these. The net effect of these aspects of control was to produce a rather tightly-coupled environment in which resources often were idle waiting for work assignments and the failure of one major component often resulted in catastrophic and total system failure. The solution to this problem is to force a condition of very loose coupling on both the logical/control decision-making process as well as the physical linkages of components. This property of "universal" loose coupling results in an environment in which the various components are required to operate in an autonomous manner.

If a single design principle must be identified as the most important or central theme of FDPS design, it is component autonomy or "cooperative autonomy" as described below. All of the other features of the definition of Fully Distributed Processing Systems given below have resulted from determining what is required to support and utilize the autonomous operation of the very loosely-coupled physical and logical resources.

2.2 The Definition of an FDPS

Fully Distributed Processing Systems (FDPS) were first defined by Enslow in 1976 [Ensl78] although the designation "fully" was not added until 1978 when it became necessary to clearly distinguish this class of distributed processing from the many others being presented. An FDPS is distinguished by the following characteristics:

1. Multiplicity of resources: an FDPS is composed of a multiplicity of general-purpose resources (e.g., hardware and software processors that can be freely assigned on a short-term basis to various system tasks as required; shared data bases, etc.).
2. Component interconnection: the active components in the FDPS are physically interconnected by a communications network(s) that utilizes two-party, cooperative protocols to control the physical transfer of data (i.e., loose physical coupling).
3. Unity of control: the executive control of an FDPS must define and support a unified set of policies (i.e., rules) governing the operation and utilization or control of all physical and logical resources.
4. System transparency: users must be able to request services by generic names, not being aware of their physical location or even the fact that there may be multiple copies of the resources present. (System transparency is designed to aid rather than inhibit and, therefore, can be overridden. A user who is concerned about the performance of a particular application can provide system specific information in order to aid in the formulation of management control decisions.)
5. Component autonomy: both the logical and physical components of an FDPS should interact in a manner described as "cooperative autonomy" [Clar80, Ensl78]. This means that the components operate in an autonomous fashion requiring cooperation among processes for the exchange of information as well as for the provision of services. In a cooperatively autonomous control environment, the components are afforded the ability to refuse requests for services, whether they be execution of a process or the use of a file. This could result in anarchy except for the fact that all components adhere to a common set of system utilization and management policies expressed by the philosophy of the executive control.

2.2.1 Discussion of the Definitional Criteria

In order for a system to qualify as being fully distributed it must possess all five of the criteria presented in this definition.

2.2.1.1 Multiple Resources and Their Utilization

The requirement for resource multiplicity concerns the assignable resources that a system provides. Therefore, the type of resources requiring replication depends on the purpose of a system. For example, a distributed system designed to perform real-time computing for air traffic control requires a multiplicity of special-purpose air traffic control processors and display terminals. It is not required that replicated resources be exactly homogenous, however, they must be capable of providing the same services.

In addition to this multiplicity, it is also required that the system resources be dynamically reconfigurable to respond to a component failure(s). This reconfiguration must occur within a "short" period of time so as to maintain the functional capabilities of the overall system without affecting the operation of components not directly involved. Under normal operation the system must be able to dynamically assign its tasks to components distributed throughout the system.

The extent to which resources are replicated can vary from those systems where none are replicated (not a fully distributed system) to systems where all assignable resources are replicated. In addition, the number of copies of a particular resource can vary depending on the system and type of resource. In general, the greater the degree of replication, particularly of resources in high demand, the greater the potential for attaining benefits such as increased performance (response time and throughput), availability, reliability, and flexibility [Ensl78].

2.2.1.2 Component Interconnection and Communication

The extent of physical distribution of resources in distributed systems can vary from the length of connection between components on a single integrated chip to the distance between two computers connected through an international network. In addition, interconnection organizations can vary from a single bus to a complex mesh network. Since a component in a distributed system communicates with other components through its own logical process, all physical and logical resources can be thought of as processes, and interactions between resources can be referred to as interprocess communication [Dav179]. For example, an application program interacting with processors and data files is accomplished through communication between logical processes.

Both the physical and logical coupling of the system components are characterized as "extremely loose." "Gated" or "master-slave" control of physical transfer is not allowed. Communication, i.e., the physical transfer of messages, is accomplished by the active cooperation of both the sender and addressees. The primary requirement of the intercommunication subsystem is that it support a two-party cooperative protocol. This is essential to enable the system's resources to exist in cooperative autonomy at the physical level.

The advantages of using a message-based (loosely-coupled) communication system with a two-party cooperative protocol include reliability, availability, and extensibility. The disadvantage is the additional overhead of message processing incurred to support this method of communication. There are a variety of interconnection organizations and communication techniques that can be used to support a message-based system with a two-party cooperative protocol.

2.2.1.3 Unity of Control

In a fully distributed data processing system, individual processors will each have their own local operating systems, which may or may not be unique, that control local resources. As a result, control is distributed throughout the system to components that operate autonomously. However, to gain the benefits of distributed processing it is required that the autonomous components of the system cooperate with each other to achieve the overall objectives of the system. To insure this, the concept of a high-level operating system was created to integrate and unify, at least conceptually, the decentralized control of the system.

A high-level operating system is essential to successfully implementing a distributed processing system. This operating system is not a centralized block of code with strong hierarchical control over the system, but rather it is a well-defined set of policies governing the integrated operation of the system as a whole. To insure reliable and flexible operation of the system, these policies should be implemented with minimal binding to any of the system's components [Ensl78].

What policies are required and how they should be implemented depends greatly on the system. For example, if it is a general-purpose system supporting interactive users, then a command interpreter and a user control language will be required to make the system's components compatible and transparent to the user.

2.2.1.4 Transparency of System Control

The high-level operating system also provides the user with his interface to the distributed system. As a result, the user is accessing the system as a whole rather than just a host computer in the network.

In order to increase the effectiveness of the distributed system, the actual system is made transparent, and the user is presented with a virtual machine and a simplified command language to access it. The user uses this language to request services by name and does not have to specify the specific server to be used. Clearly, the same request might be assigned a different server depending on the state of the total system when the request is made. However, to make the system truly effective for all users, knowledgeable individuals must be able to interact with the system more intimately, requesting specific servers or developing service routines to increase the efficiency or effectiveness of the system [Ensl78].

2.2.1.5 Cooperative Autonomy

Cooperative autonomy has already been described at the physical interconnection level. It is also required that all resources be autonomous at the logical control level. That is, a resource must have full control of itself in determining which requests it will service and what future operations it will perform. However, a resource must also cooperate with other resources by operating according to the policies of the high-level operating system. Cooperative autonomy is an essential prerequisite for systems to have fault tolerance and high degrees of extensibility [Ensl78]. It is perhaps the most important as well as the most distinguishing characteristic of a fully distributed processing system.

2.2.2 Effects on System Organization

Although the detailed design of the hardware and software required to implement an FDPS is still in progress, it has been possible for some time to identify certain characteristics that these components must have. One area in which certain criteria already appear reasonably well defined is the nature of the organization of the following system components:

- Hardware
- System control software
- Data bases

It should be noted that a number of definitions and descriptions of distributed systems in general are based on the principle that one or more of these components is physically distributed. (Some such discussions add to this list a fourth component --- "processing or function;" however, considering the distribution of processing independent from the distribution hardware is quite improper. Why distribute the hardware if it will not have some function to perform; similarly, how can the processing be distributed without a corresponding distribution of the hardware? That would be processing on a truly "virtual machine.")

An important characteristic of an FDPS is that, in order to meet the definitional criteria given above while also attempting to provide as many as possible of the benefits listed in Table 1, all of the three components listed above must be physically distributed and the degree of distribution must in each case exceed a reasonably well-defined threshold. A diagram illustrating this requirement is shown in Figure 1. The various organizations of each component, identified and positioned along each axis, is not meant to be an exhaustive list. These points are listed to better identify the relative location of the three thresholds defining the volume of space occupied by FDPS's. (It might also be noted that it seems quite proper to characterize any system that is not in the "origin cube" as being "distributed" to some degree.)

2.2.3 Some Excluded Systems

Considerable work has been done on new system designs to achieve subsets of these benefits, but very few systems have made substantial progress toward meeting all of the criteria. Perhaps the most widely known of these is Arpanet; however, only the communication subsystem of that network qualifies in this respect. Many other systems, some of which are discussed elsewhere in this magazine, have made substantial improvements in subsets of the areas of system performance; examples are the Honeywell Experimental Distributed Processor, the Cm* system at Carnegie-Mellon University, Mininet at the University of Waterloo and ICOPS at Brown University. However, the number of systems mislabeled as distributed data processing systems far exceeds these.

Most of the criteria contained within the definition are met by crossing a threshold on a particular dimension. The definition is not a set of binary criteria, and better understanding of these criteria and their thresholds can be obtained by considering some systems that are excluded by the definition.

It excludes, for example, distribution within a single mainframe. One writer has characterized the architecture of several of the modern processor systems that include independent I/O channels as "incorporating distributed processors since (it) contains separate I/O processors, arithmetic logic processors and possibly diagnostic processors." [Gild76]. Such a categorization has little utility and has not found very wide acceptance. Obviously, there is a permanent binding of tasks to the various components in this type of system organization.

A front-end processor that controls communication with a mainframe definitely does not constitute the type of distributed system defined here. Although it may meet some of our criteria, it also is dedicated to one function and is not freely assignable.

Many instances of a master/slave relationship occur in both hardware and software control. The key point is that the recipient of the information transferred, be it data or a control signal, cannot decide whether or not to accept the transfer and act upon it. When this concept is implemented in hardware, it is often referred to as gated transfer. In software control systems the master/slave relationship is quite commonly encountered in multiple computer and basic multiprocessor operating systems.

The continued decline in the price of hardware has made more and more attractive new multiple-processor system organizations incorporating specialized functional units, such as vector multiplier, a floating-point arithmetic unit, or a fast Fourier transform unit. In the general concept of operation, such dedicated function processing is only slightly different from a master/slave relationship. The major difference is that the master/slave control relationship also excludes many hardware systems containing multiple general-purpose processing units from our definition. What causes some of the terminology confusion with these configurations is that these specialized services are often provided by a general-purpose unit, such as a programmable microprocessor. The functional unit may be "specialized" by a microprogram, or it may be completely general but utilized in a dedicated functional role, such as a minicomputer to control input/output in a larger system. The distinguishing characteristic of this class of excluded systems is the dedication of the resource to a single or a fixed set of functions. It operates in a master/slave mode, as far as the control over its own activities is concerned. The criteria of both free assignment and autonomy are violated.

There is wide agreement (except perhaps among marketing and advertising people) that a single host processor with a collection of remote terminals that simply collect and transmit data does not qualify as a distributed data processing system, even if the terminals are intelligent and do some editing and formatting.

Even the presence of multiple hosts in a complex network interconnection structure does not necessarily make the system distributed. It may be distributed from the point of view of switching; but from the point of view of overall operations and control, it usually is centralized. Systems such as these do not have the capability for dynamic reallocation or reassignment of tasks in the event of hardware failure.

Intelligent terminals systems are most often presented as distributed processing systems in advertising copy. However, the operation of a system with intelligent terminals or local processors has to be studied carefully to determine to what extent the processing is actually distributed. Such a system (several are commercially available) consists of several terminals connected to a local processor that has secondary storage capabilities, such as disks or cassettes. It offers intelligent data-entry or field-editing and similar functions executed in the local processor through the execution of a program stored there. It has shared file access, but only to local files. It communicates with a main processor, but to do so, the local processor must emulate a "dumb" terminal in order to use normal protocols. Finally, it is capable of remote job entry. There is no indication of any distribution of the control function, for the distribution of work is fixed and a local terminal cannot affect it.

A terminal with a resident text editor, whether it is provided by hardware or software, is not an example of a distributed data processing system. In order to meet the definition, their terminal must be "smart" enough, first, to do some real work, and second, to recognize when it cannot accomplish its assigned work and to pass it on to another appropriate service unit. The simple off-loading of work to a higher level when this level is fully utilized is just the beginning of the transition to fully distributed processing. If the terminal coordinates several concurrent and simultaneous remote jobs, giving each a different type of service at a different location, without human intervention, then it more closely resembles a distributed system. The threshold is reached when the local control system can decide whether work should be done locally or passed on to the rest of the system, basing its decision on an analysis of local workloads and capabilities. Distributed processing is definitely not equated with merely "moving equipment to the periphery of a business system to capture and process data at the source."

Perhaps the intelligent terminal does have a role to play in the development of distributed processing systems. It may facilitate a painless transition to more decentralized organizations for hardware and data storage as well as control. This is accomplished by adding features to the local system and making other modifications that increase the local functions, prior to establishing higher-level system connections and a complete build-up of global functions.

2.3 High-level Operating Systems

The high-level operating system is a key ingredient in the distributed data processing system. Its design must take into account several characteristics and problems.

The classical design for operating systems, as it has developed, assumes the availability of a large amount of system information. Although the completeness and validity of information about the work being presented by the user is questionable, the operating system is usually assumed to have access to complete and accurate information about the environment in which it is functioning. This is not the case in a distributed data processing system; complete information about the system will never be available. The resources provide a service, but they may either intentionally or unintentionally, shield information from outside inspection.

In distributed systems, there will always be a time delay in the collection of information about the status of the system components. The ramifications of these time delays are extremely important. In a conventional centralized processor, the operating system can request status information, being assured that the interrogated component will not change state while awaiting a decision based on that status information, since only the single operating system asking the question may give commands. In a distributed data processing system, the time lags that occur can become significant; as a result, inaccurate (badly out-of-date) information can be transmitted because the autonomous component proceeds along its own path. If you have ever worked with input/output device handlers, you surely have wondered whether or not the information that has been obtained is accurate. For distributed data processing systems, it will be essential to raise the degree of paranoia of the system designer to a much higher level than for centralized systems. The system must be designed to work even with erroneous or inaccurate status information.

A further complication with regard to system information available is the possibility of variations in the information presented to different system controllers. These variations may be a result both of time delays and of differences in the shielding of information from different controllers. As LeLann has observed, "This absence of uniqueness, both in time and in space, has very important consequences." [LeL77].

2.4 General Control Problems

High-level operating systems as described here are highly nonhierarchical -- that is, they are single-level and have no internal master/slave relationships. This characteristic, combined with component autonomy, greatly exacerbates the control problems. Even if autonomous multiple components are cooperating, the probability of simultaneous conflicting actions is much higher than in hierarchical systems. Also, synchronizing the actions of the various controllers in the system is much more difficult, because of the presence of appreciable time-lags. Finally, the problem of deadlocks or infinite cycles within the system is quite different from that associated with other systems. Some proposals call for an umpire (an outside third party) to solve this problem; however, such an umpire would have to be transient, since the presence of a permanent umpire would denote an unacceptable degree of hierarchical control.

From the operating characteristics of the distributed processing system, some conclusions can be drawn about the nature of system communication. The second criterion of our definition requires a message-type protocol for all transfers, both physical and logical, both in interprocess communications and interprocessor communications. There must be no global variables and there must be no tunneling across system components. All parameters must be passed across well-defined and rigidly enforced interfaces.

Much of the work done on communication in uniprocessor and multiprocessor environments is applicable, but extensions to the solutions found there are definitely required to cope with the autonomous nature of the system components in the distributed system.

The user must communicate with the system by directive containing service names only. Our criterion of system transparency makes unnecessary and perhaps impossible to the user designation of the system component offering a desired service. However, this requirement introduces new problems of system failure and user error detection, since no one processor can establish whether the service requested can be provided anywhere in the system, or even whether it is legal.

Resource management in a distributed processing system is a multidimensional job. Thus far, very little work has been done on the aspects of resource management that apply specifically to distributed processing systems. However, low-level functions are quite similar to those performed on uniprocessors; they include physical resource allocation, and management of those facilities required by a process after it has been scheduled on a particular system component. Before that can be done, however, the required resources may have to be assembled at one location, or linkage mechanisms established so that they can be used remotely. The problems that have to be addressed in that process are locating the resources, determining which components are suitable, and determining the best way to move the resources to the selected location. At an even higher level is the scheduling problem, determining when a function should be initiated or terminated.

Any system exhibiting monolithic, autonomous control presents completely new problems in system scheduling. A request for service in a nonhierarchical system might well result in an initial denial of that service by all physical resources. In that instance, the requesting entity might initiate an evaluation of relative priorities between the new request and currently executing tasks, followed perhaps by bidding (priority adjustments) and preemption. The efficient execution of this procedure is one of the most important functions of the high-level operating system.

When all of these problems and their possible solutions are compared to similar problems and solutions encountered in uniprocessor systems, the major factor exacerbating the distributed system control problem is seen as communication within the distributed data processing system, which is asynchronous with respect to the detailed execution of the functions, and which exhibits time-lags in addition to the communication processing time itself. Uniprocessors cope with many of the problems with semaphores, flags, lockout gates, or timeouts. To attempt to do this in a reasonably complex distributed system requires too much time, in the sense that such practices greatly reduce the throughput rate of the system. Bear in mind that transit time for signals transmitting the semaphores is on the order of 100 milliseconds. In addition to the lowering of performance, the reliability and the robustness of most of the uniprocessor solutions are in doubt, since a system operation such as TEST-and-SET cannot be replicated as a single indivisible machine-level instruction that can be executed immediately on the next machine cycle.

The problem of time is further complicated by the fact that most of the procedures, such as voting and software synchronization, which have been presented as solutions to the difficulties introduced by transit time, require even more processing by every component in the system.

2.5 Programming Languages for Distributed Systems

Four aspects of distributed processing systems have a significant impact on the goals of a language design effort. First, data is stored throughout the system in a distribution which is in some sense natural (for example, data may be stored where it is generated or it may be stored where it is easily accessible to those who use it most frequently). Second, it may be infeasible to move data from node to node for processing. Third, a single application may need to access data that is stored on a number of different nodes. Fourth, a programmer should not need knowledge of where data is stored in order to access it.

It should be noted that fully distributed processing does not necessarily require new programming languages, much less new models on which to base programming languages. Any program can conceivably be run on a distributed system; however, when a program needs to access data on multiple nodes, a single thread of execution is unlikely to be executed efficiently. Furthermore, even languages with parallel execution features are not adequate in a fully distributed environment. The key issue is that most programming languages have not been designed to allow a programmer to provide information about the nature of program execution or to describe the appropriate structural units of the program needed by an operating system in order to make effective allocation and scheduling decisions in such an environment. Thus, a major goal is to design language features that will elicit information and provide structural units which will simplify allocation and scheduling decisions. Our other major goal is that in doing so, the language should present a natural and helpful framework for the description of a large class of programs.

The most important aspect of our initial design work is the model of computation on which our language will be based. In order to explain the motivations for our computational model, we need to take a closer look at fully distributed systems. Conceptually, a fully distributed system consists of a number of independent machines (where a 'machine' may denote one or more processors) with communication links between them. Each machine has a processing capability, a storage capability and a message handling capability. Furthermore, each machine functions with a large degree of autonomy (the system as a whole may make requests of the individual machines but has no control over how these requests are carried out) and there is no memory shared between the machines.

In order to achieve our design goals, our computational model mimics the logical structure of a fully distributed system. In this model, a program consists of a number of execution modules (which can even be thought of as individual programs) and a network description. In other words, a program is a network of execution modules. The execution modules are independent and contain local variable declarations (there is no shared memory), port declarations and executable code. (Ports are used to communicate with other execution modules and permanent storage facilities.) Port to port connections, port to permanent storage connections and the execution of the execution modules are controlled by the network description. Thus, the execution modules (because they resemble the machines of a fully distributed system) are structural units which will simplify allocation and scheduling decisions. On the other hand, the network description contains information which will enable an effective allocation and scheduling of these units.

Two aspects of our model should be helpful in the writing, debugging and reading of programs. First, because execution modules are independent (sharing only communication links and a common network description), they may be developed, tested and understood separately. Second, because all of the control and network communication specifications are contained within it, the network description

provides a meaningful abstract view of the program as a whole. This control and communication abstraction should contribute significantly to the understandability of distributed programs and perhaps even to programs written for existing system organizations (not meeting our criteria for 'fully' distributed).

2.5.1 Research Issues in 'Distributed' Language Design

Communication primitives in languages for distributed computing are one of the most important issues that should be addressed by the participants at the workshop. The most obvious alternatives include message-based communication and call-based communication (for example, the rendezvous in Ada). However, the potential for the network description presented above to be an active program unit opens up new possibilities. It could function as a communication controller for its execution modules, providing any hybrid communication primitives desired by a programmer. The utility of such an approach requires further examination. Due to our interest in very loosely coupled systems, messages are the most likely candidate for our language currently being developed. However, for systems where loose coupling is not a dominant consideration, use of communication primitives implemented by a network controller might prove quite useful.

Another important issue raised by the model of programs presented above is how distributed programs are to be described and controlled. A program might quite reasonably be composed of execution modules compiled and stored at different nodes in a network, conceivably composed of heterogeneous processors, and perhaps even written in different languages. These questions lead into a number of subproblems: conventions for naming files throughout a distributed system, interactions between programming languages and command languages (note that our network descriptions fall somewhere between the traditional roles for these two languages), and primitives needed by a programmer for the control and coordination of multiprocess execution.

3 CONCLUSIONS

The concepts of distributed data processing clearly hold a great deal of promise for solving many of the problems and limitations currently faced by system designers. It is important, however, to make a critical analysis of the operational characteristics of any system that is addressing those issues. This paper has reported on such effort --- The Georgia Institute of Technology Research Program in Fully Distributed Processing Systems.

4 ACKNOWLEDGEMENTS

The work discussed in this paper was performed as part of the Georgia Institute of Technology Research Program in Fully Distributed Processing Systems. Twelve faculty members have been involved in this major research program as well as four staff members and over thirty students. The program has been supported by a number of agencies with the principle funding being provided by the Office of Naval Research, U.S. Navy, under contract N00014-79-C-0873.

5. REFERENCES

- Dav179 Davies, D. W., Barber, D. L. A., Price, W. L., and Solomonides, C. M., Computer Networks and Their Protocols, John Wiley and Sons, 1979.
- Ensl74 Enslow, Philip H. Jr. (ed.), Multiprocessors and Parallel Processing, New York: John Wiley and Sons, 1974.
- Ensl78 Enslow, Philip H. Jr., "What is a 'Distributed' Data Processing System?" Computer (January, 1978): 15-21.
- Ensl81 Enslow, Philip H. Jr., "Distributed Data Processing --- What Is It?," AGARD Avionics Panel Symposium on "Tactical Airborne Distributed Computing and Networks," Norway, (June 22-26, 1981).
- Gild77 Gilder, Jules H., "Distributed Processing: Keyword for Tomorrow's Supercomputers," Computer Magazine, (April, 1976): 14.
- Hopp79 Hopper, K., Kugler, H. J., and Unger, C., "Abstract Machines Modelling Network Control Systems," Operating Systems Review 13 (January, 1979): 10-24.
- Lein58 Leiner, A. L., and Weinberger, A., "PILOT, the NBS Multicomputer System," Proceedings of the Eastern Joint Computer Conference (1958): 71-75.
- LeLan77 LeLann, Gerard, "Distributed Systems--Towards a Formal Approach," IEIP Congress Proceedings (1977): 155-160.

Table 1. "Benefits" Provided by Distributed Processing Systems

A Representative List Assembled from Claims Made in
Actual Sales Literature

High Availability and Reliability
Reduced Network Costs
High System Performance
Fast Response Time
High Throughput
Graceful Degradation, Fail-soft
Ease of Modular and Incremental Growth
Configuration Flexibility
Automatic Load and Resource Sharing
Easily Adaptable to Changes in Workload
Incremental Replacement and/or Upgrade
Easy Expansion in Capacity and/or Function
Good Response to Temporary Overloads

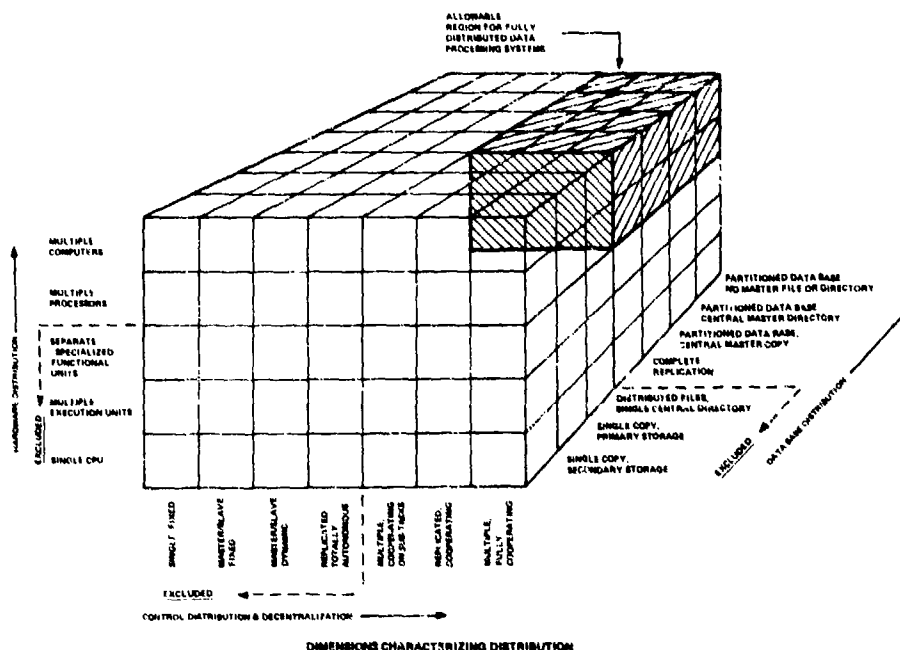


Figure 1. Axes of Distribution

THE EFFECT OF INCREASINGLY MORE COMPLEX AIRCRAFT AND AVIONICS

ON THE METHOD OF SYSTEM DESIGN

J.T. MARTIN

FERRANTI COMPUTER SYSTEMS LIMITED
Western Road, Bracknell, Berkshire, England.

SUMMARY

This paper describes the evolution of Aircraft and their associated 'Avionics'. The evolutionary progress is considered as starting from a simple low speed Aircraft with rudimentary flight instruments and sighting systems, through the interconnection of some of these systems and progressing to the recent Avionic Systems with Centralized Digital Computing.

The paper shows how the changes in aircraft systems, from the simple analog connection of a few systems, through the analog sensor - interface box - centralized digital system, to the sensor producing digital outputs - interface box - centralized digital system, have produced comparatively small changes in the methodology used for the design of these systems.

The move to systems containing distributed processing interconnected by digital highways is shown to be revolutionary rather than evolutionary and to require a new approach to the System Design problem so as to reap the maximum advantage from the available computing capability.

1. INTRODUCTION

The first aircraft used in war were seen not as fighting vehicles but as information gatherers, especially for the artillery, indeed the ability of the aircraft to fight was scorned by the Generals who controlled them and, in the beginning, ignored by those who designed them. The main pre-occupation being the production of as stable a platform as possible.

Aerial warfare started in two ways. For air to air attack the standard service revolver was used and for air to surface attack standard army grenades were simply thrown from the cockpit. No attack avionics were involved, the sighting systems being the barrel of the hand held revolver or the pilot's impression of his position over the target. It is from these rudimentary beginnings, less than 70 years ago, that today's highly sophisticated fighting aircraft have evolved. Today we have aircraft specifically designed for either air to air or air to surface attack, aircraft where the cost of the attack avionics approaches that of the airframe itself and on which the attack avionics seeks to integrate information from most of the available aircraft systems and sensors.

The following sections of this paper will describe this evolution in slightly more detail and show how the System Design process has, up to now, been modified only slightly in order to cope with the increased complexity.

2. THE EVOLUTION OF AIR TO AIR ATTACK

It is hardly surprising that pilots engaging in air to air combat utilising service revolvers or rifles rarely succeeded in shooting down their targets. The frustration engendered by this failure of their air to air weapon system together with a sudden realization by those in charge that if aircraft were useful to them in an observation role then they must be equally as useful to the opposition, and should perhaps be deterred, led to the requirement for a more effective weapon system.

The more effective weapon system became machine guns either loosely mounted on the aircraft and aimed by an observer using a sight fixed to the gun or a machine gun rigidly mounted on the aircraft and aimed by the pilot aiming the whole aircraft, and hence the gun, utilizing a simple ring and bead sight mounted on the aircraft.

Although a spectacular increase in success rate was achieved by these methods it was clear that further improvements could be made. However, this was how the 1914-1918 air war was conducted.

The simple ring and bead sight suffered from two main disadvantages, firstly the parallax error inherent in attempting to track a target with a mechanical sight some inches from the eyes created large errors, secondly firing at a moving target, the opposing aircraft, means that the bullets must be fired not at the target but at the position that the target will be at when the bullets arrive.

These sources of error were reduced by the use of the graticular sight which both removed the source of parallax error and allowed some degree of 'aiming off' although the accuracy of this latter process depended to a high degree on the quality of the pilot's estimation of the speed and attitude of the target in relationship to his own aircraft.

The first real sign of avionics in gunsights did not appear until the experimental Gyroscopic Lead Computing Optical Gunsight appeared in 1940, entering service as the GGS Mk. 2 in 1942. This sight used a gyroscopic sensing unit and enabled an automatic computation of the required lead angle based on a measure of the rate of turn of the sight line (measured by the gyroscope), the range (estimated from the pilot's appreciation of the target size and the velocity of the bullet or shell (fed into the gunsight as a design parameter).

The 'avionics' associated with the above gunsight was in fact amazingly simple, consisting of a gyroscope to enable the computation of lead angle and non-linear variable resistors to allow the pilot to enter range and thus the gravity fall to be expected to be experienced by the bullets or shells during their trajectory. However, for the first time an aircraft sensor, albeit one specially added into the gunsight, was being used in the air to air weapon system.

Range of target was still being entered into the system from a subjective appreciation supplied by the pilot. In 1949 this requirement on the pilot to provide such information was removed by the advent of radar ranging systems whereby a target range supplied by radar returns could be inserted directly into the gunsight.

From this point systems quickly emerged whereby, as the target was held and tracked on the radar and the necessary adjustments for lead angle compensation and relative position of target and attacking aircraft were carried out by electronic computation, the pilot no longer had to be able to see his target in order to engage it.

At this stage the air to air attack system could be considered to have been produced. Aircraft sensors supplying information regarding the motion of the attacking aircraft coupled with the sensor information (from the radar) on the target, produce for a pilot, who may not even be able to see his intended target, the necessary cues to enable his attack to be promulgated.

3. The Evolution of Air to Surface Attack

In the same way that the hand held revolver proved fairly ineffective for air to air attack, the hand thrown bomb also proved to be somewhat less than perfect. There were two main reasons for this lack of effectiveness, firstly it was somewhat unlikely that the pilot or observer would accurately hit the intended target, having nothing better to calculate a release point with than his judgement of the track of the aircraft and an impression of the drop characteristic of the weapon. Secondly the fact that a bomb that is capable of being picked up and thrown from the cockpit is somewhat limited in size and thus effectiveness upon reaching the ground.

The second of these problems was comparatively easy to solve - larger bombs attached to the aircraft which fall off upon the application of some form of release command.

The first problem, that of producing the release pulse at the correct time is not quite so simple. The path followed by the bomb as it falls will basically depend on the flight characteristic of the weapon, the velocity of the aircraft (and hence initial velocity of the weapon) at the point of release and the distance that the weapon must fall (the height of the aircraft at release). Thus occurs the same type of progression as for gunsights, we move from the simple mechanical sights of early aircraft to the complex release point calculating computers which are supplied with target position (from radar or laser seeker), aircraft height (from altimeter), ground speed, air speed and aircraft track and heading (from air data computers).

4. OTHER SYSTEMS

So far only the comparatively simple problem of gunsights and bombsights have been considered and, although these are undeniably important parts of any aircraft weapon systems, it is evidently apparent that there is no point in having these systems if the aircraft cannot be positioned in the right place at the right time so as to be able to use them. It is thus worthwhile to consider some of the other major system components required.

4.1 Navigation System

Early aircraft had very limited ranges and speeds. In the case of the artillery observation aircraft it could often see its own trenches and navigated by flying from one visual landmark to another, landing in a convenient field if it did lose its way.

With increasing range, speed and landing weight it became necessary to be able to navigate to a target or area and back to a somewhat more prepared landing strip than the nearest convenient field. Initially it was possible to achieve this objective by continuing to use the visual landmark with perhaps a good idea of in which direction the sun should be. Two factors spoil this happy state of affairs. Firstly the improved performance and gunsights of the fighter aircraft decreed that the bomber should fly at night and secondly somebody decided that flying should not be solely a fair weather occupation.

Thus began the two main methods of aircraft navigation - radio aids and aircraft position dead reckoning.

Radio aids for navigation have progressed from the simple bearing from a controller enabling a returning fighter to be vectored back to its base, through the radio highways produced for bombers by such systems as Knickerbocker, Wotan and Oboe to the sophisticated position fixes supplied by systems using Omega and eventually Navstar. Amongst these radio aids can also be counted the ground mapping radars introduced in the 1939-45 war and progressively improved ever since.

The dead reckoning aids include such systems as integrating air data computers and of course Inertial Navigation systems whose accuracies increase almost yearly.

4.2 Communication Systems

Communications both between aircraft and the ground and between aircraft have progressed somewhat in the last 65 or so years. We have moved from the artillery aircraft's different coloured Very lights and the pilot's arm indicating a potential target to another pilot, through the radio with channels A to E, to the complex array of VHF, UH and HF channels available to a modern pilot and his crew.

It is now possible for the pilot or crew of one aircraft to select a target for attack and for that target to be automatically indicated to the pilot or crew of another aircraft via a digital data link completely automatically and without a word having been spoken. (As an aside it is also interesting to note that with the advent of JTIDS we have reverted to the line of sight range of the original Very signal, but at least the information rate has been increased).

4.3 Pilot's Aids

From the above very brief summary of the advances made in aircraft weapon and supporting systems it can easily be seen that the work load of the pilot or crew of the modern aircraft has increased enormously over that enjoyed by his historical counterpart. If we add to this list such systems as ESM, ECM, ECCM, the fact that the aircraft is now flying faster, that the aircraft is probably the target of attacking missiles, that it is firing missiles, that air engagements between aircraft may be measured in periods of seconds we can very quickly see that the pilot or crew can do with any help that they can get.

Avionics can, if used intelligently, solve some of these problems which, to a certain extent, it has helped to create. Computers can be used to select that information which the pilot needs to know, cathode ray tubes can be used to display connected information together, or display the most important advisory notices always in the same place and not spread around the cockpit, multifunction keyboards can replace banks of switches (some of which always inevitably seemed to end up in ergonomically bad positions) and also provide prompts as to the information or actions required from the pilot or crew. It is at this stage that System Design should commence.

5. THE AVIONIC SYSTEM AND ITS DESIGN

As can be seen from the above descriptions of the evolution of avionics on aircraft the early sub-systems gunsights, bombsights, navigation etc. did not form an overall system nor were they designed to do so. For instance, in section 2 above, we saw how when rate of turn was required to be supplied to the GGS Mk. 2 gunsight the aircraft sensor, the gyroscope, was added to the gunsight producing a self contained unit.

The next step was for one sub-system to supply to another some particular piece of information required by the recipient sub-system, range from the radar being supplied to the gunsight, for instance.

Certainly up to this point System Design was concerned with ensuring that the various sub-systems on an aircraft worked satisfactorily but the total Avionics was still a very loosely coupled collection of separate sub-systems rather than being designed as a total system. The interconnection of sub-systems was tenuous to say the least and agreement about particular interfaces between two sub-systems could be, and was, made without consideration being taken, or, to be fair, needing to be taken, of the other aircraft sub-systems. As long as the synchro outputs from one sub-system matched the orientation of the inputs to the other and the voltages produced and read at the ends of the connection were agreed as to their meaning then that was generally the end of the System Design task.

Thus were produced systems such as that shown in extremely simplified form in figure 1. Information is passed from one sub-system to another as required and as agreed by the sending and receiving parties. If transformation of the data in terms of, for instance, units was required then it was generally carried out as required for each sub-system and many different transformation of units might be carried out for one particular piece of data depending upon which sub-system it was being sent to or received by - the transformation being carried out by the sub-system least unable to cope with the additional work.

During the 1960s digital computers became available to carry out some of the computations necessary within the respective sub-systems, however, due to their size and cost they could not effectively be added to any sub-system that required to carry out a calculation. This led to the concept of a small number of centralized (from the system point of view) computers receiving data from sensors or sub-systems, carrying out the necessary calculations, and then feeding the results to the sub-systems that required the results. Unfortunately as the interfaces to the sensors and sub-systems still tended to be analog in nature and as the units used by one sub-system were unlikely to match those required by another a large part of the centralized computer task was taken up by performing analog to digital and digital to analog conversions and in performing digitally the necessary furlongs per fortnight to knots unit conversions.

Figure 2 shows an, again extremely simplified, example of such a system. Unfortunately the system design techniques used for the loosely coupled system described above still tended to be used for the production of this type of system. Sensors and sub-systems produced those parameters which were demanded of them and demanded those parameters which they needed. Both sets of parameters being produced or demanded in the format and units most easily handled by the sub-system, with the resultant interconnection tangle being left to be sorted out by the centralized computer and its associated interface adapters. Thus the computer ended up as being the go between for any two potentially connecting systems rather than the producer of a unified total system.

The next stage in the evolution of avionics produced some sub-system or sensors containing digital inputs or outputs instead of the older analog interfaces. This allowed some of the analog to digital and digital to analog adaptors to be replaced by digital to digital adaptors, a not very encouraging step. The problem, of course, was one of standardization. Every manufacturer or project had its own pet digital interface and somehow it was always the incompatible ones that were trying to get together.

6. TOWARDS THE 'PERFECT' SYSTEM

During the 1970s two very important things happened. The digital computer became both small and affordable and Mil. Std. 1553 was created and won a large measure of acceptability.

Given a digital interface enjoying widespread acceptance the interface adaptors could be put aside. More importantly, given a digital bus, all sub-systems are automatically connected to all other sub-systems requiring intercommunication without having to worry about the complexity or cost of producing a special link for an information path which although desirable is on the face of it not positively essential.

In the same way, given that most sub-systems and sensors can now be, and are now being, supplied with their own digital computational facilities it is possible for these sub-systems to supply the information that is required, in the format that is required, to the sub-systems that require it.

Suddenly information can be supplied, in the correct format, as and when required. Many of the old system design constraints have been removed, reversion can be supplied not by the old back up sub-system approach but by re-configuring the system data flow. The processing power limitation of the old centralized computer can be forgotten, but how do we design the system. It is no longer possible for two sub-system designers to come to a gentleman's agreement about the data to be passed between them in terms of format and repetition rate. Now every piece of data produced by one sub-system is potential information for every other sub-system.

With a system of the type as shown in figure 3 the main limitation is the ability of the System Design Methodology to cope with the design of the system not of the individual sub-systems to cope with their tasks.

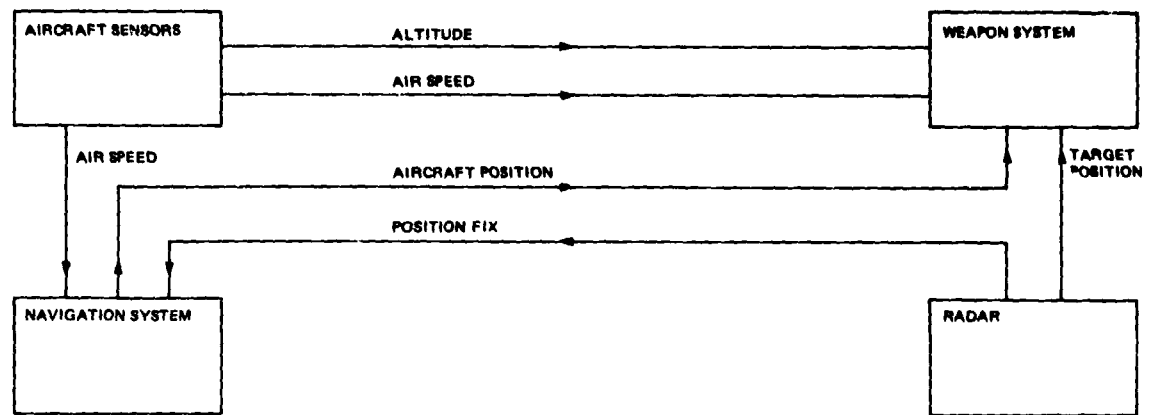
7. CONCLUSION

It is now possible, perhaps for the first time ever, to fully integrate an Avionic system and to provide a means whereby all the necessary, rather than essential, information paths can be provided.

We saw, in the example of the gunsight, for instance, how in the past sub-systems have been connected together so as to provide only the essential information required within the sub-system but in isolation to the remainder of the total system. Even with the advent of the centralised computer, whether connected to the remainder of the system by analog converters or discrete digital links, the total system has tended to be made up of a number of sub-systems with the computer acting as the interface device between sub-systems and serving the needs of connected sub-systems rather than providing an overall integrated system.

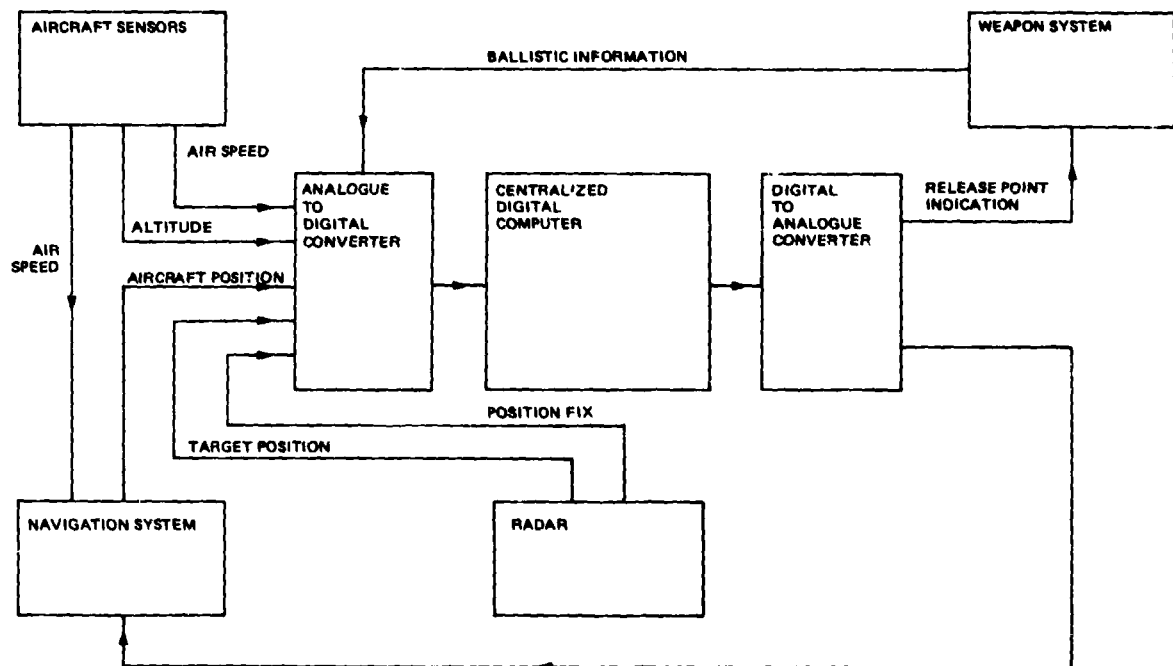
Throughout this period the task of system design has been that of producing compatible interfaces between one sub-system and another and attempting to produce, with these collections of sub-systems, a final product that approximates to the original requirements of the customer and intentions of the designer. Given the facts of both distributed computing and sub-system interconnected by a common highway, this rather simplistic (although often far from simple) approach to system design can no longer cope with the problem to be handled.

To reap the advantages that can be gained from a system built using today's available technology requires that the system design task must be commenced from the viewpoint of the customer's requirement and then broken down into the sub-systems required to produce the end result. It is no longer possible to arbitrarily assign tasks to sub-systems without having considered the effect of the assignment on the total system. The methods used for system design must be able to cope with the task of designing the total system as a unit rather than a collection of sub-systems. It is only in this way that full advantage can be taken of the computing power that is potentially available in the modern avionics system.



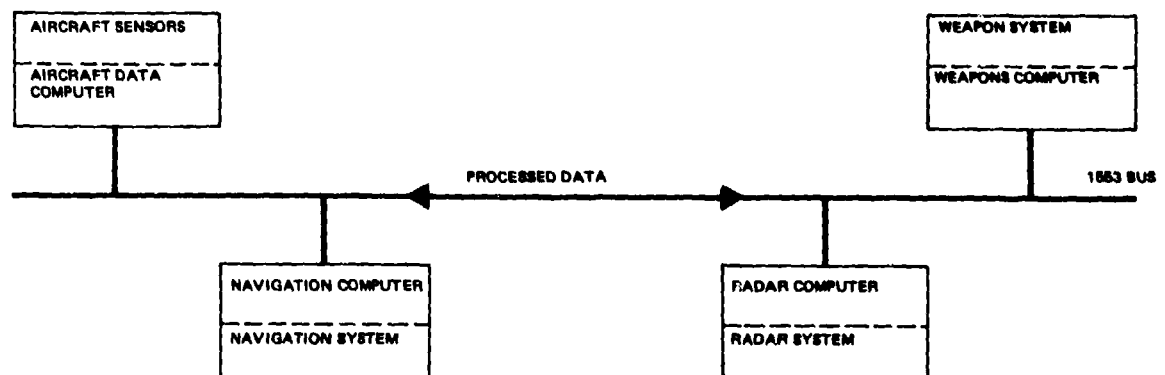
Example of Loosely Coupled System

Figure 1



Example of Early Centralized Computer System

Figure 2



Example of Integrated, Distributed Processing System

Figure 3

A TUTORIAL ON DISTRIBUTED PROCESSING IN AIRCRAFT/AVIONICS APPLICATIONS

BERNARD A. ZEMPOLICH
DEPUTY TECHNOLOGY ADMINISTRATOR FOR COMMAND, CONTROL AND GUIDANCE
RESEARCH AND TECHNOLOGY GROUP
NAVAL AIR SYSTEMS COMMAND, WASHINGTON, D. C. 20361

SUMMARY

The purpose of this tutorial is to present an overview of the state-of-the-art in real-time distributed processing as applied to aircraft/avionics. Definitions and concepts are presented starting with the total aircraft as a real-time distributed computer-controlled system. The relationship of aircraft mission and avionics system architectures is discussed. Overall system architectural considerations are identified and their impact upon a Real-Time Distributed Computer-Controlled System is detailed. A top-down hierarchical, architectural structure is presented. This top-down structuring is described in terms of the logical functional decomposition of the system as follows: total aircraft/avionics system partitioning of aircraft/avionics subsystems, interconnect bus structure (network), system-wide processing architecture, subsystems definition, and computer systems.

1.0 INTRODUCTION

By the early 1960s, operational needs in combination with the need for on-board equipment flexibility led to the introduction of general purpose, programmable digital computers into a variety of aircraft/avionics systems. The programmability of these machines permitted rapid operational and technical changes to be made through software modifications rather than through hardware changes. The advent of the integrated circuit also hastened the introduction of general-purpose digital computers because of the weight and volume savings that these electronic devices had over other competing technologies. These "first generation airborne computers" were termed "centralized"; that is, all Operational Flight Programs were contained in the memory of a single machine. Unfortunately, while computer hardware made great strides forward in the state-of-the-art during this period in time, the associated software tools did not. Thus, while the use of digital computers allowed the introduction of many new operational capabilities, management also had to live with costly, highly complex, and in many instances, inefficient use of the computer as an operational resource due to the (then) lack of quality software development and support tools.

As the solid-state electronics technology matured, and its products applied to militarized computers, the physical characteristics of the on-board computers decreased in value, which, in turn, led to the availability of a number of light-weight, lower cost computers. The availability of these computers led to their incorporation (physically) into various on-board subsystems. Thus, the term "embedded computers" came about. And eventually, these machines were connected together in what was subsequently termed a "federation" of computer resources.

As time progressed, the introduction of general-purpose, programmable digital computers continued to bring about quantum improvements in operational capabilities to military aircraft. Unfortunately, due to the (then) lack of computer hardware standards, these machines were individually unique from both hardware and software support considerations. Furthermore, this situation was exacerbated by the fact that the solid-state electronics industry continued to introduce microelectronic circuits with greater densities, higher speed performance, and myriad circuit types which made obsolete almost overnight, technology advancements which had not yet been fully operationally utilized in a military environment.

The continuation of proliferation of hardware, the absence of suitable standards, and the ever-increasing speed at which new solid-state electronic devices were being invented and/or created and subsequently manufactured, led to the establishment by the late 1970s of standards for computer hardware and related higher order languages. As a generalization, it can be stated that this is the technical management situation which exists today in 1981.

As we entered the decade of the 1980s, there were many questions yet to be answered relative to computer architecture and language standards. Specifically, it was postulated that the decade of the 1980s and 1990s would see the introduction of Real-Time Computer-Controlled, Aircraft/Avionics Distributed Systems containing several hundred microprocessors interconnected by various digital bus schemes. These microprocessors would be embedded throughout the aircraft as computer resources which control the operation of a highly fault-tolerant, reconfigurable, hierarchically structured aircraft/avionics system.

A major technical management challenge facing the avionics community today is how to transition from the current inventory of analog "black boxes" to one in which by the 1990s the inventory will be approximately 90% digital in nature. The main reason why this is a major challenge to the avionics community is that throughout this transition period, it is imperative to maintain hardware interchangeability and not upset, nor negate, established hardware and software standards. Table 1 identifies the key technical characteristics in aircraft/avionics equipments by time frames. The time period 1980 to 1990 itemizes characteristics expected to be the foundation for future Aircraft/Avionics Real-Time, Distributed Computer-Controlled Systems.

TABLE 1

<u>1940 - 60</u> <u>ANALOG</u>	<u>1960 - 80</u> <u>CENTRAL DIGITAL</u>	<u>1980 - 2000</u> <u>DISTRIBUTED DIGITAL</u>
<ul style="list-style-type: none"> • Wired programs • Dedicated analog processors • Integration through pilot displays • No redundancy • Limited fault tolerance • No dynamic reconfiguration capability • Discrete hardware 	<ul style="list-style-type: none"> • Stored computer program • Central processor(s) • Communication through I/O integration and central processor/stored program • Some degree of redundancy • Some degree of fault-tolerance • No dynamic reconfiguration capability • Use of MSI & LSI hardware 	<ul style="list-style-type: none"> • Distributed hierarchical stored program • Redundant central processor(s) • Distributed, dedicated functional processors • Communication through a bus network • Large scale use of multi-path redundancy • Fault-tolerance and dynamic reconfiguration • VHSIC hardware

2.0 SYSTEM ARCHITECTURAL STRUCTURES

As the avionics community entered the decade of the 1980s, there was an absence of a generally accepted system architectural approach to the design and development of on-board aircraft/avionics equipments and systems. In the absence of a formal system architectural definition, a "Pseudo-Hierarchical Architectural Structuring" is proposed (see Table 2). This concept is designated as "pseudo" solely because of the current lack of "reduction to practice" (implementation) of such an approach. It should be noted, however, that the top-down decomposition of the system architectural structure is real from an engineering design viewpoint and does indeed lend itself to a logical, natural methodology for decomposition of a system into its constituent parts.

TABLE 2

SYSTEM PSEUDO-HIERARCHIAL ARCHITECTURAL STRUCTURING

- Total aircraft/avionics system
- Partitioning of aircraft/avionics subsystems
- Inter-connect bus structure
- System-wide processing architecture
- Subsystems definition
- Computer systems

The total aircraft/avionics system is presented as being at the top of the Pseudo-Hierarchical Architectural Structuring as shown in Table 2. It is presented as the equivalent of the system mission for the aircraft. The system mission is presented for definition purposes as being the operational functions performed by the aircraft such as: fighter, attack, Anti-Submarine Warfare (ASW), Airborne Early Warning (AEW), cargo and/or passenger, or Electronic Warfare (EW). It is the system mission which determines the types, capabilities, functions, and performance of the various aircraft/avionic electrical and electronic equipment on-board the aircraft.

2.1 PARTITIONING OF AIRCRAFT/AVIONICS SUBSYSTEMS

The on-board subsystems required for any given aircraft system mission can be partitioned into a number of groups of equipments which perform a general functional purpose. For example, the Vehicle Group of subsystems would contain such equipments as the flight controls, pilots' displays, and the electrical generators. The Core Avionics Group would contain the communications, navigation, and the computational resources. The Mission/Sensors Group would contain the specific radars, acoustic sensors, or the electronic warfare hardware equipments. The Weapons Group is of course self-explanatory as to its contents. It should be noted that these four major partitions or groups of subsystems are "glued" together by the System Architecture, Integration, and Common Hardware considerations.

For the foreseeable future, it would appear that the interconnect bus structure will continue to be based upon the requirements of MIL-STD-1553. However, this statement is not meant to imply that the technology of implementation will necessarily remain the same. It is logical to expect that as a minimum, a fiber-optics bus will be fully implemented and operational by 1990.

2.2 SYSTEM-WIDE PROCESSING ARCHITECTURAL ALTERNATIVES

Figure 1 is a "road map" of the various System-Wide Processing Architectural Alternatives available to designers and developers of future aircraft/avionic systems. It would seem reasonable to assume that

more and more the Distributed or Federated Control approaches will be used in future aircraft designs, while the Centralized Control approach would more likely continue to appear in technology updates of current aircraft systems. For definitional purposes, System-Wide Processing Architectures are defined by this author as consisting of all of the on-board embedded computer resources: hardware, software, and firmware.

As stated previously, there is an absence of formalized industry and government approaches to aircraft/avionic system architectural considerations. Thus, the following definitions of various processing architectures are provided as working definitions only. That is, they are possibly subject to refinement and/or modification. The definitions of Centralized, Distributed, Federated, and Hierarchical System-Wide Processing Architectures are presented in Table 3 in terms of their key hardware and software characteristics.

TABLE 3
SYSTEM-WIDE PROCESSING ARCHITECTURES CHARACTERISTICS

<u>Hardware Characteristics</u>	<u>CENTRALIZED</u>	<u>Software Characteristics</u>
<ul style="list-style-type: none"> • Powerful central computer (may use multiprocessor or redundant computer) • All communication through central unit • Other computers look like peripherals 		<ul style="list-style-type: none"> • Single complex executive resident in central unit • Application programs cover all avionics system functions • Central unit provides all systems control
	<u>DISTRIBUTED</u>	
<ul style="list-style-type: none"> • High speed computer/computer bus • Reconfiguration not difficult if: <ol style="list-style-type: none"> 1. All computers have same architecture 2. Multipath communication with peripherals 		<ul style="list-style-type: none"> • Low complexity local executives in each computer • Applications programs limited to local functions in any other computer • No single source for system control (system control distributed throughout local executives)
	<u>FEDERATED</u>	
<ul style="list-style-type: none"> • Hardware tailored to function • Low data bus rate communication bus treated like peripheral • Reconfiguration difficult • Computers may have different architectures 		<ul style="list-style-type: none"> • Single executive resident in one unit • Application programs limited to local functions in any one computer • One unit provides general system control
	<u>HIERARCHICAL</u>	
<ul style="list-style-type: none"> • High speed computer/computer global bus • Low speed local bus for control • Computers have same architecture but tailored capability • Reconfiguration not difficult 		<ul style="list-style-type: none"> • Global bus system looks distributed • Local bus systems look federated with global bus interface computer acting as executive

Figures 2, 3, 4, and 5 provide diagrammatic representations of the four major System-Wide Processing Architectures previously identified in Table 3. Based on the working definitions given in Table 3, architectural options available for consideration by military aircraft/avionic system designers are listed in Table 4 and are diagrammed in Figures 6 through 11. These options offer the aircraft/avionic system designers the latitude to maximize those characteristics which are of major importance to the particular aircraft/avionic system application.

TABLE 4
SYSTEM ARCHITECTURAL OPTIONS APPLIED TO MILITARY NEEDS

- Option 1: Full Functional Redundancy (Figure 6)
- Option 2: Full Functional Redundancy Plus Dedicated Subsystems (Figure 7)
- Option 3: Maximum Physical Redundancy (Figure 8)
- Option 4: Full Functional Redundancy Within Local Group of Subsystems (Figure 9)
- Option 5: Centralized (Figure 10)
- Option 6: Multiprocessor (Figure 11)

2.3 STANDARDIZATION - ARCHITECTURE INTERACTION

Figure 12 is an attempt to visually demonstrate the inter-relationships between computer software and hardware resources and the System Architecture, Integration, and Common Hardware requirements. It is hoped that the need for simultaneous consideration of all of these factors by system designers can be explicitly visualized from the structure of the matrix.

In Figure 12, the FOUNDATION for the entire system is shown as the horizontal bar entitled "System Architectures". Being that it is a foundation, it cuts across each of the vertical bars which are meant to convey the idea that the "Missions" are independent, separable, and unique to each operational mission need. Contained within this concept of the System Architecture as the foundation upon which all the operational systems are built, is the premise that any item identified within the block has general applicability to all military aircraft systems (when required).

The horizontal bars listed under "Common Functions" are used to indicate equipments or software which cut across various Missions, but are uniquely tailored to the particular operational application. For example, signal processors and their associated software programs are used in many Naval aircraft; however, it is only for the Anti-Submarine Warfare (ASW) Mission that the processor and its associated software are tailored for the acoustic processing role. In like fashion, the aircraft displays may have some identical hardware and software used across all aircraft, but again, any one particular combination of controls and displays is unique to each Mission application.

2.4 DISTRIBUTED EMBEDDED COMPUTATIONAL RESOURCES

A key indicator of the degree of distribution of embedded computational resources within an aircraft/avionic system architecture is the total number of microprocessors used within the on-board equipments. Shown in Table 5 are projected number counts for "futuristic" Airborne Early Warning (AEW) and an Anti-Submarine Warfare (ASW) aircraft. The information contained in this chart was prepared by a major supplier of navy aircraft, and as such represents, in the author's opinion, a fairly realistic projection of the quantities of microprocessors that will be used as on-board embedded computer resources with the next generation of naval aircraft. Worthy of particular note is the fact that the count difference between the two aircraft operational applications lies in the area of Mission Avionics rather than in the Core or Vehicle Systems areas.

TABLE 5

TOTAL SYSTEM MICROPROCESSOR COUNT

AEW APPLICATION (137 Microprocessors)		ASW (141 Microprocessors)	
Mission Avionics	- 24	Mission Avionics	- 28
Core Aircraft Systems	- 16	Core Aircraft Systems	- 16
Core Avionics	- 97	Core Avionics	- 97

Table 6 itemizes the number of reprogrammable and fixed program microprocessors projected for certain types of functional avionic equipments. This chart was prepared by a firm currently engaged in providing similar equipment for operational use. And again, as with the statement made relative to the information contained in Table 5, it reflects more than a reasonable degree of engineering certainty as to the validity of the estimates shown.

TABLE 6

SELECTED AVIONICS SUBSYSTEMS MICROPROCESSOR COUNT

Function	Number of Reprogrammable Microprocessors	Number of Fixed Program Microprocessors	Total
Data System and Displays (Core Mission)	25	64	89
Core Sensors & Conditioning	16	26	42
Acoustic Signal Processing	8	12	20
Radar Signal Processing	6	6	12
Other ASW Sensors & Conditioning	9	26	35
TOTAL	64	134	198

3.0 FUTURE TECHNOLOGY CONSIDERATIONS

There are a number of considerations which must be taken into account relative to the transfer and insertion of new technologies into future Real-Time Aircraft/Avionic Distributed Computer Control Systems. First of all, the Real-Time, Computer-Controlled, Distributed System of the future will require that the system conceptual and definition phase of each future aircraft program consider the inter-relationships of factors such as: support/tools software, applications software, firmware, computer-aided design, test and manufacturing software, processing system architecture software, and simulation, test and diagnostics software.

Secondly, system designers and developers must take into consideration the technology directions listed in Table 7.

TABLE 7

TECHNOLOGY DIRECTIONS

- Software function taken-over by firmware in near-term
- VHSIC chips take over software functions in the long-term
- Emergence of hardware macros as basic building blocks
- Signal processing as dominant thrust
- More "Modular" software
- New systems will be fault-tolerant, redundant, reconfigurable
- Emergence of the "Smart", reconfigurable memory system
- Greater thrust for extracting data from aircraft/avionic systems.

Lastly, the systems designers and developers must consider the items listed in Table 8. These items represent the author's best judgment as to the challenges to be faced by the management and engineering staffs both in government and in industry involved in the system conception, definition, design, development, test and evaluation, and subsequent logistic support of future Real-Time, Computer-Controlled Distributed Systems for aircraft/avionic applications.

TABLE 8

CHALLENGES TO BE FACED

- Amount of embedding into the system architecture
- Systems engineers not computer specialist/engineers performing the design function
- Primary failures will be at the system level not at the component level
- Lack of economic leverage
- Rapidity of change in the microprocessor state-of-the-art
- Fixed function vs. programmable microprocessors
- Lack of precise definitions throughout the field.

4.0 CONCLUSION

If there is any one conclusion which can be reached in trying to understand the attributes of Real-Time Aircraft/Avionic Distributed Computer Control Systems it would have to be, in the opinion of this author, that system designers and developers can no longer build such systems from the "bottom-up", black box approach. A partnership between the technical managers, the system designers, and the various technologists is required if future systems are to be developed with minimum proliferation of the embedded computer resources, minimum logistics for both the avionics hardware and the software, and maximum availability in the operational environment.

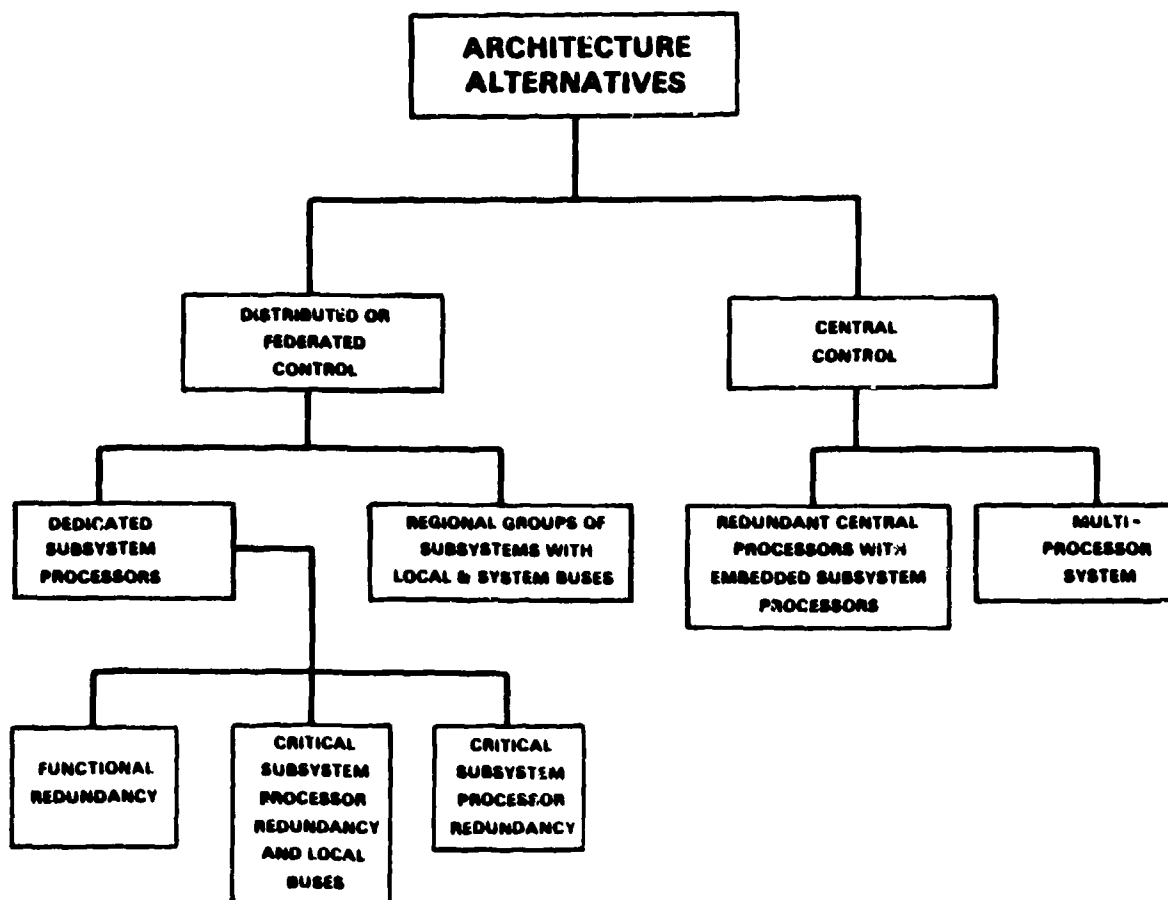


Figure 1 System-Wide Processing Architectural Alternatives

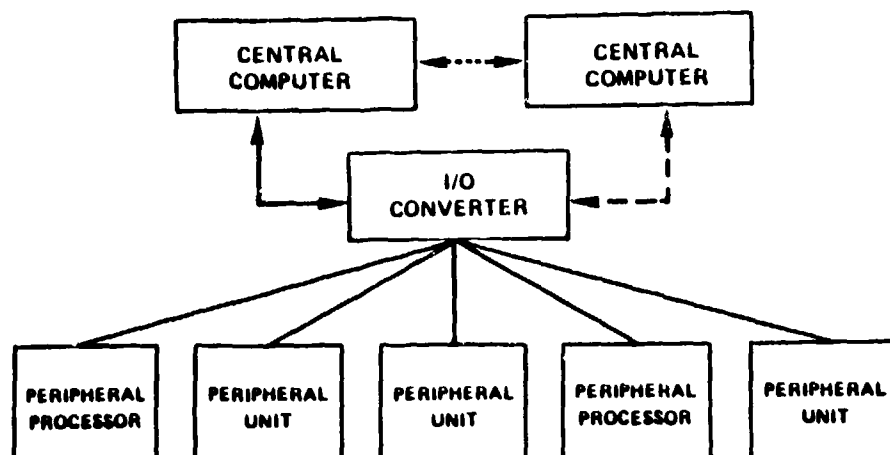


Figure 2 Centralized Architecture

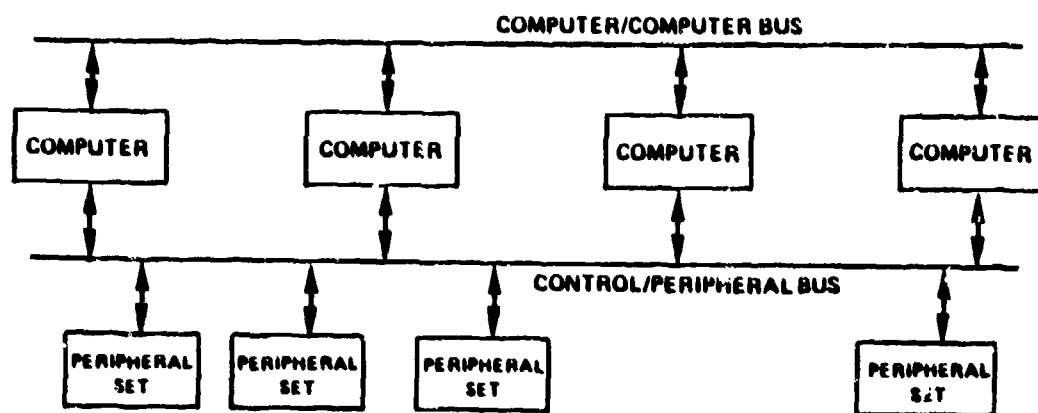


Figure 3 Distributed Architecture

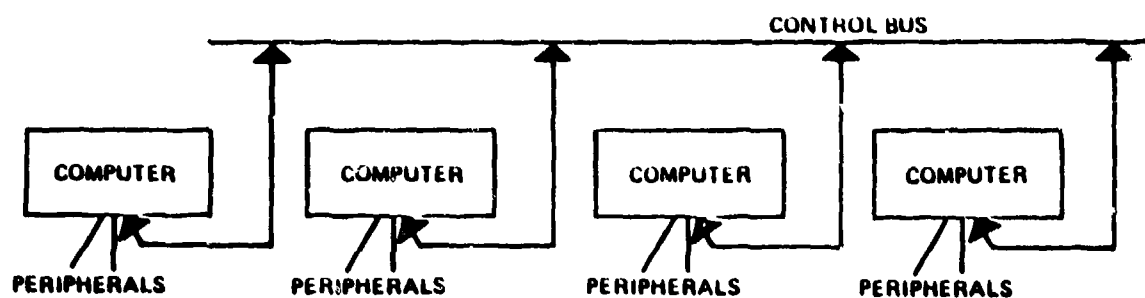


Figure 4 Federated Architecture

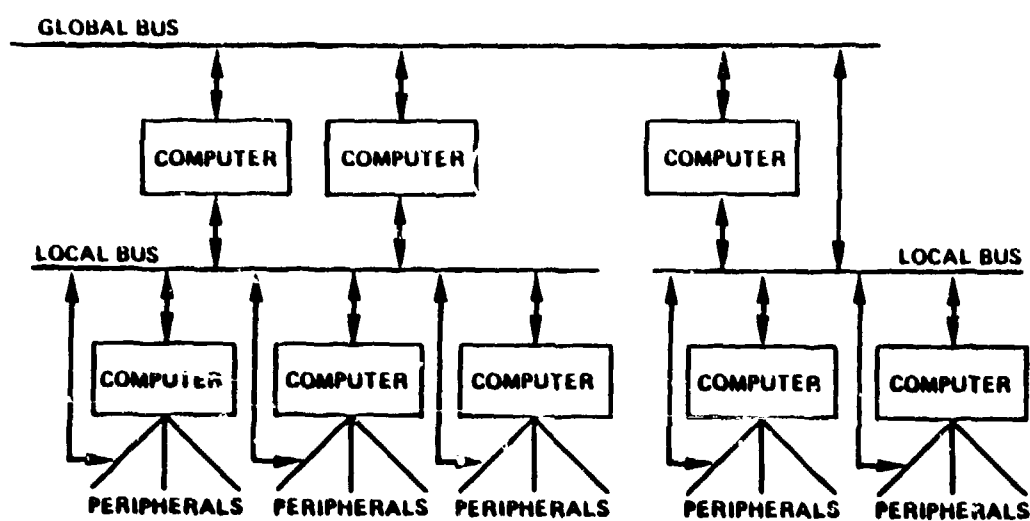


Figure 5 Hierarchical Architecture

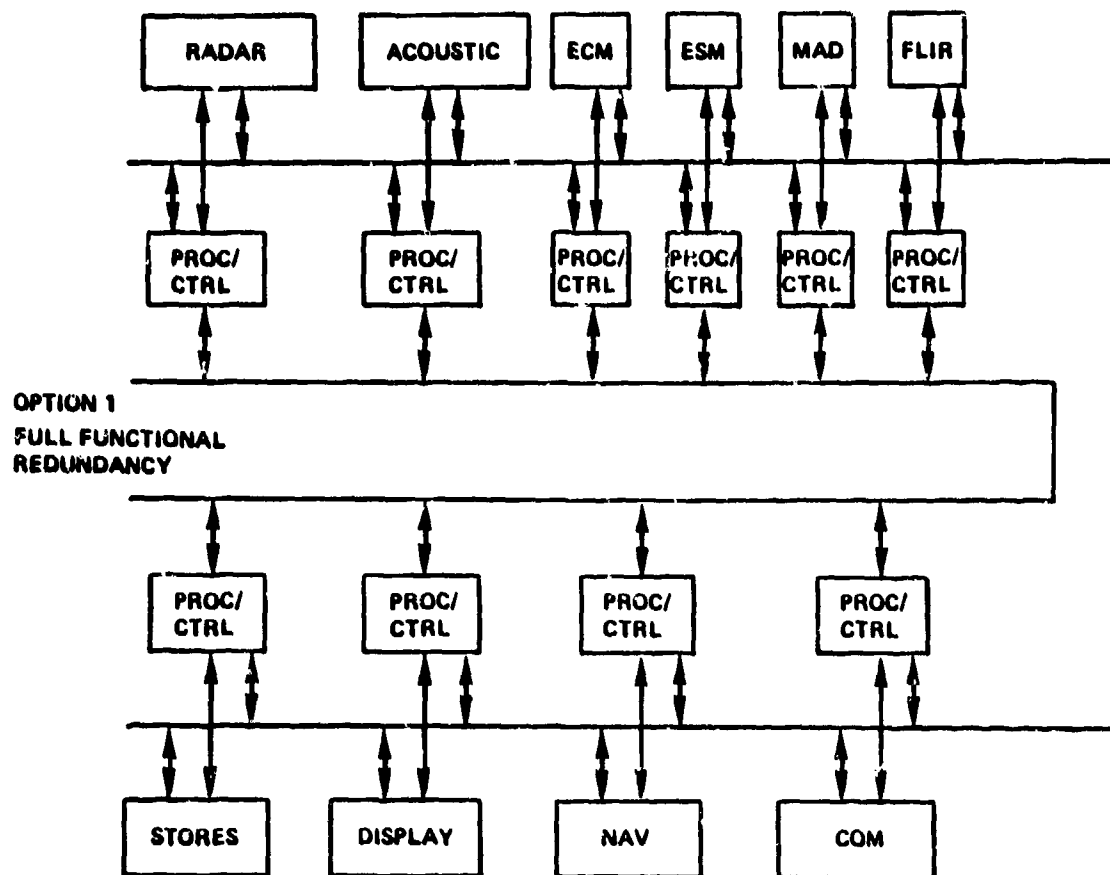


FIGURE 6 - OPTION 1

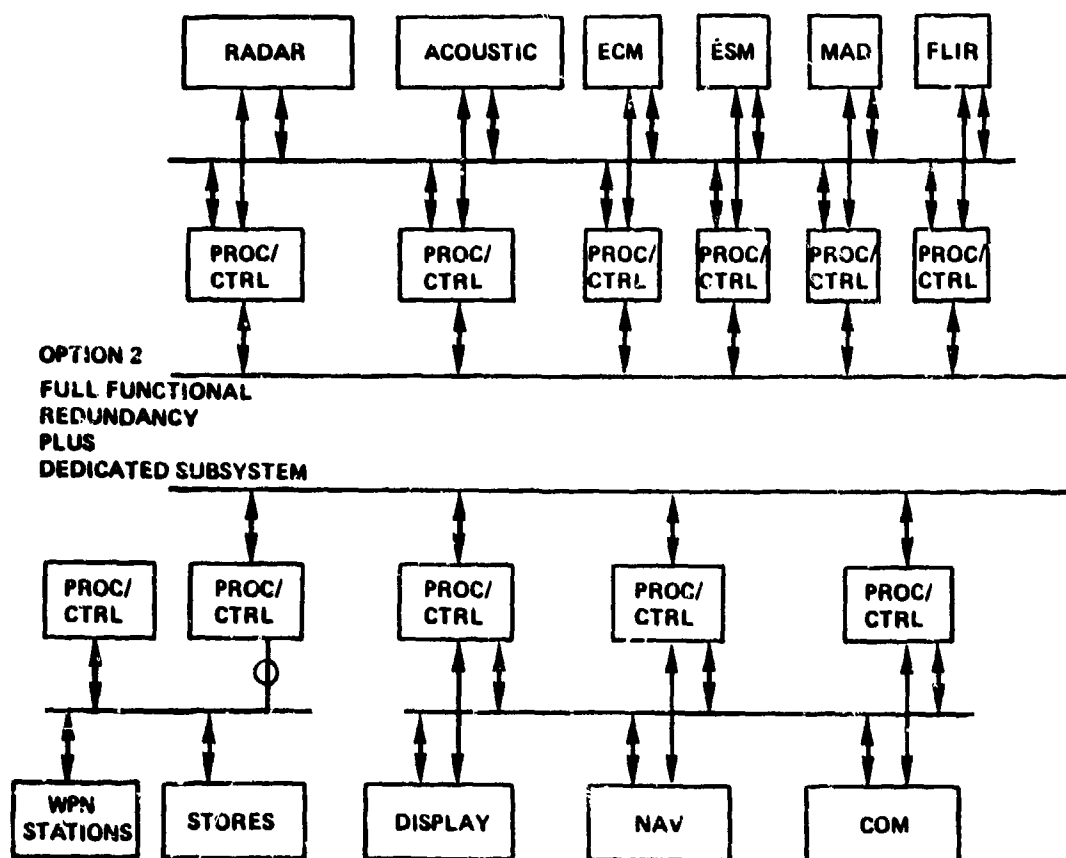


FIGURE 7 - OPTION 2

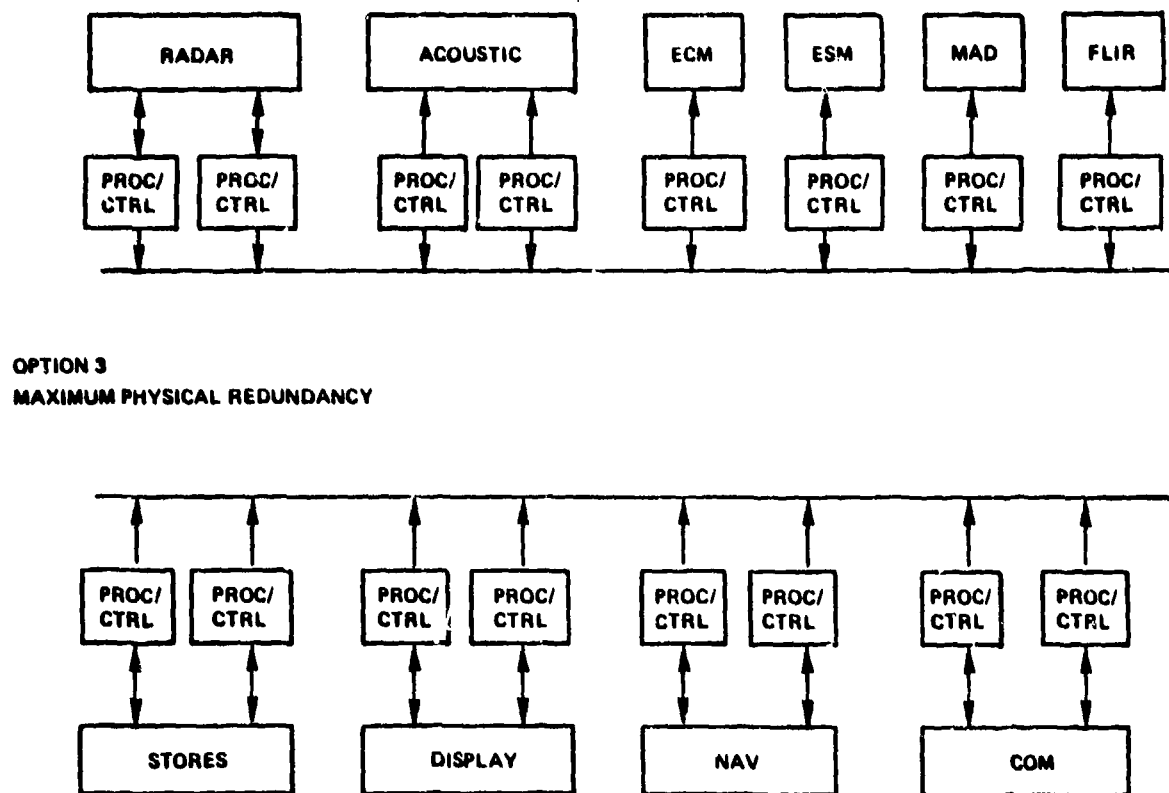


FIGURE 8 - OPTION 3

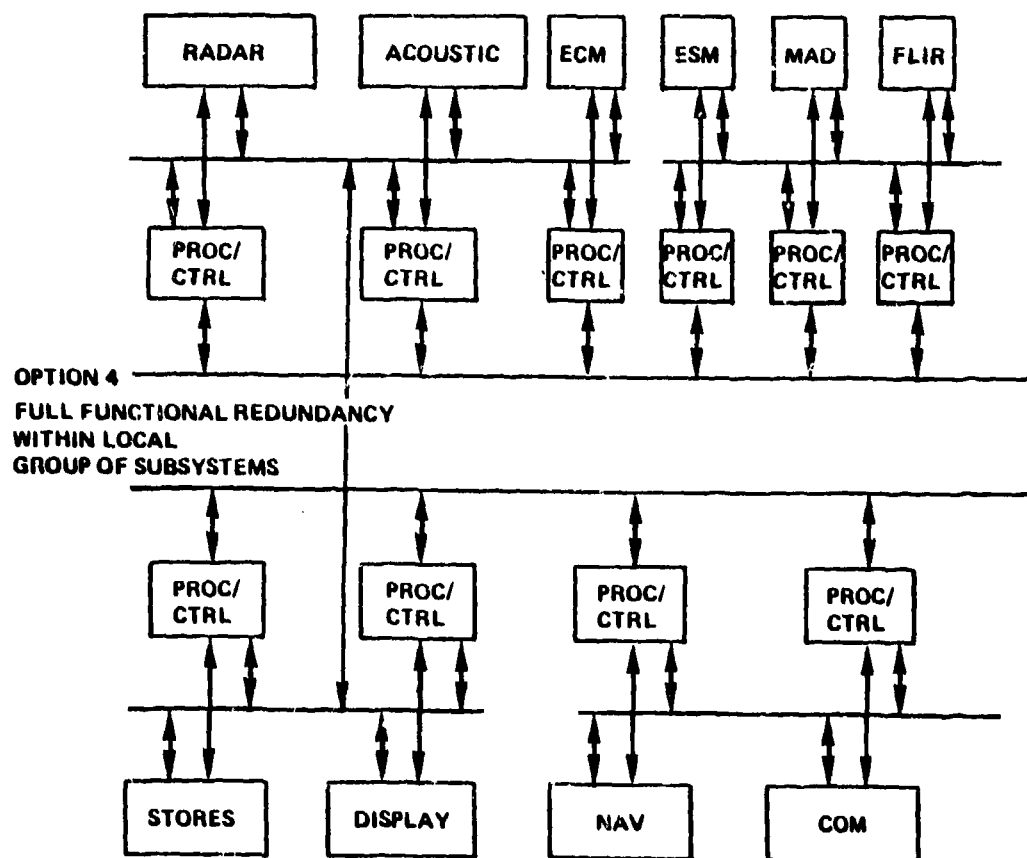


FIGURE 9 - OPTION 4

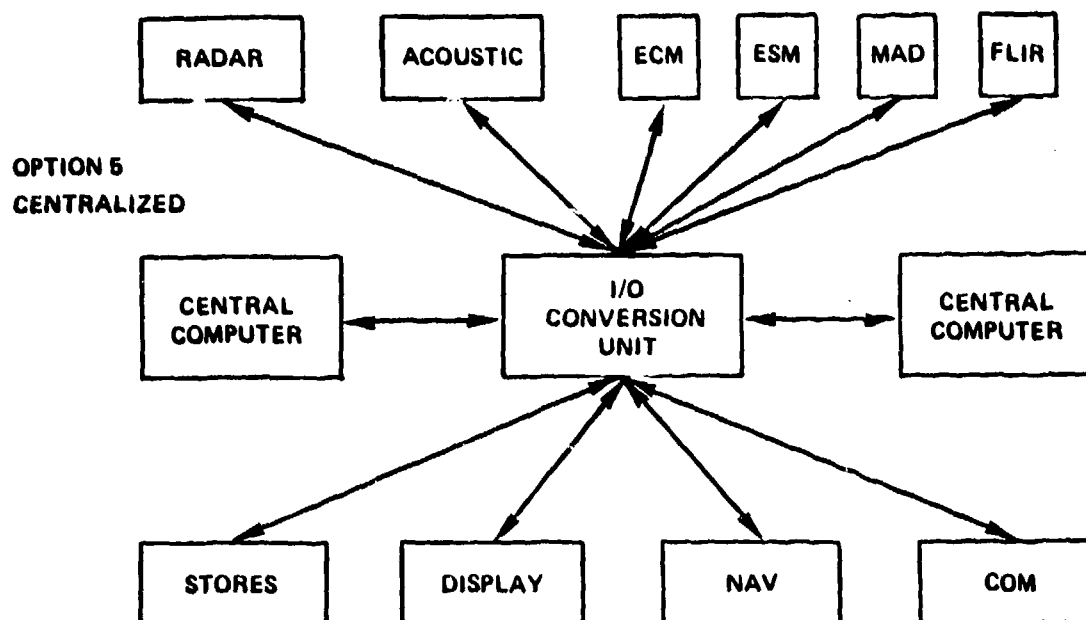


FIGURE 10
OPTION 5
CENTRALIZED

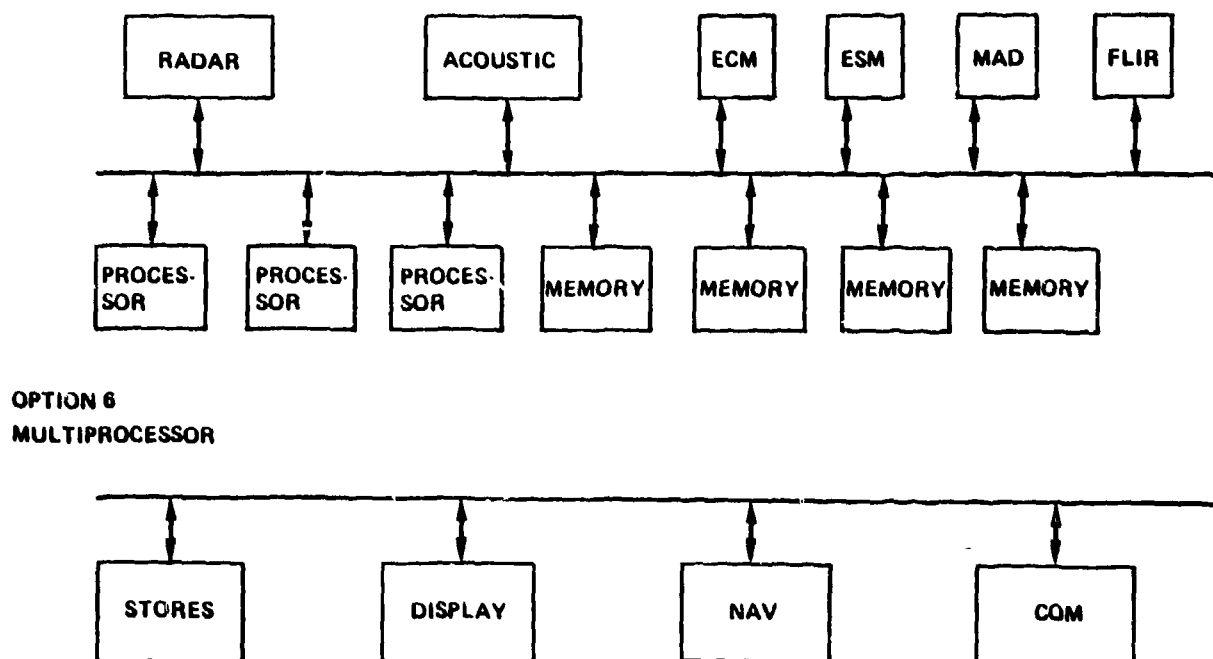


FIGURE 11
OPTION 6
MULTIPROCESSOR

STANDARDIZATION - ARCHITECTURE INTERACTION MATRIX

MISSIONS

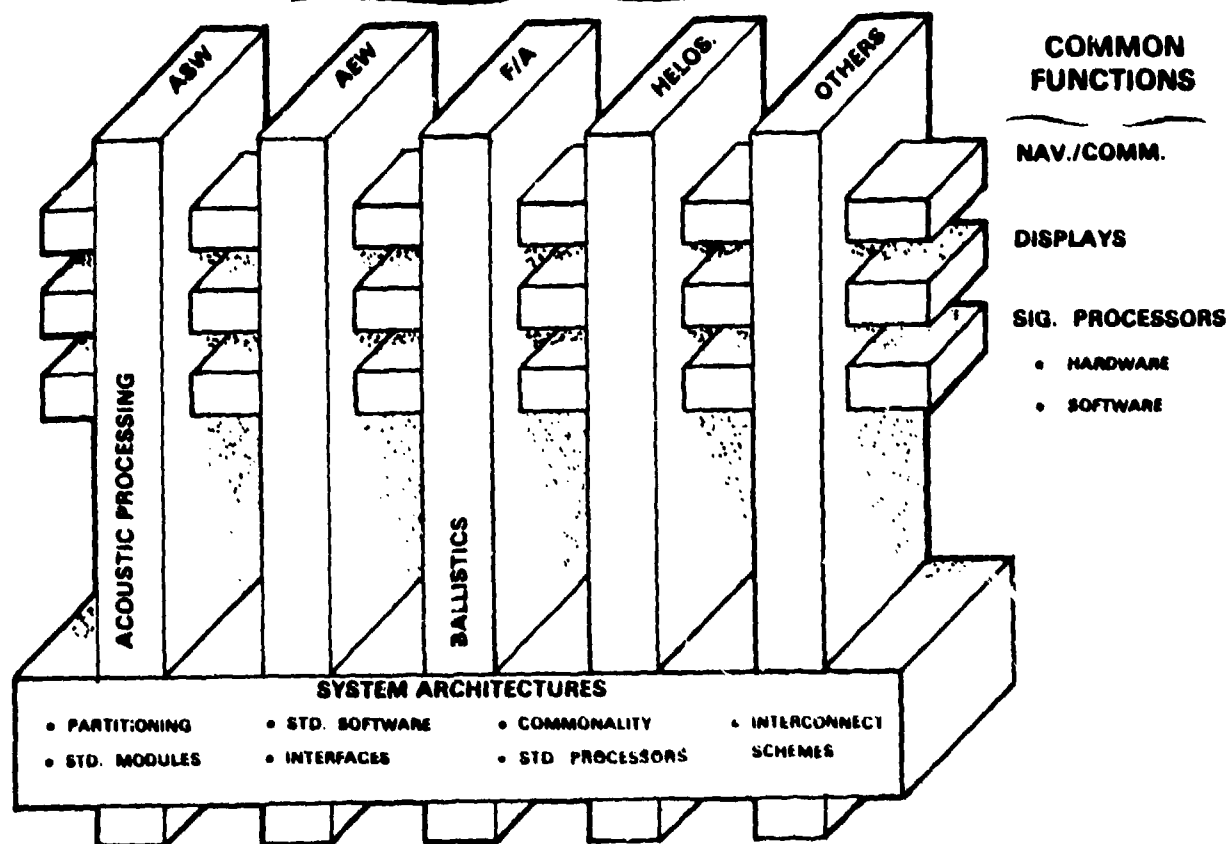


FIGURE 12

DISCUSSIONS

SESSION I

REFERENCE NO. OF PAPER: I-1

DISCUSSOR'S NAME: Dr. von Issendorff

AUTHOR'S NAME: Enslow (Livesey, presenter)

COMMENT: You mentioned that there are no suitable programming languages for distributed systems so far. Among others there are CSP and PLITS from Feldman or ADA. Could you please comment on why these languages are not sufficient?

AUTHOR'S REPLY: Sufficient for what? These languages do, of course, allow us to write distributed or concurrent programs, but this is only 10 percent of the problem. We need active programming environments including program specification, design, verification and debug tools for distributed systems (test tools, too). These problems are especially difficult in a distributed system. (This is the opinion of the presenter, and not necessarily that of the author.)

REFERENCE NO. OF PAPER: I-1

DISCUSSOR'S NAME: Erwin C. Gangl, WPAFB, Ohio

AUTHOR'S NAME: Enslow (Livesey)

COMMENT: In the application of distributed systems, flight safety concerns are reliability of hardware and software performance and guaranteed real-time response. This can be obtained by maturing the software through extensive use and correcting the bugs. We cannot use this approach in flight safety systems since we have to have reliable software prior to first flight. Therefore, we have to accomplish this through exhaustive testing. How can we get reliable software by testing since in distributed systems it is impossible to predict and exercise all possible states of the system?

AUTHOR'S REPLY: This is also true for centralized systems and is not a special problem of fully distributed processing systems. I do not know of any "magic" solutions, but rather see the continued use of top-down design, verification, automatically generated test cases, extensive simulation and perhaps new tools such as IPC control languages. (This is the opinion of the presenter and not necessarily that of the author.)

REFERENCE NO. OF PAPER: I-1

DISCUSSOR'S NAME: B. A. Zempolich

AUTHOR'S NAME: Enslow (Livesey)

COMMENT: Do you distinguish between ADA as a programming language and software development tools, such as Hardware Description Languages?

AUTHOR'S REPLY: I'm not an ADA expert. However, the direction I see ADA going is that users of ADA will subset it and that subset will look a lot like PASCAL. Another group of programmers will be trained primarily in the use of tasking facilities. Other programmers will spend most of their time on developing more advanced debugging, testing, and specification tools to fit around ADA--the so-called ADA environment. I expect the most exciting work to be done in the environment rather than language development itself. We have enough programming languages. What is needed are the tools to enable people to use them.

REFERENCE NO. OF PAPER: I-1

DISCUSSOR'S NAME: Dr. van Keuk

AUTHOR'S NAME: Enslow (Livesey)

COMMENT: Would you say that it will remain to be sensible to think about distributed processing without addressing a precise, limited, and well-analyzed case of application being in mind? The software and hardware structure will often be dictated by the particular kind of application to a high degree.

AUTHOR'S REPLY: I think that both jobs are needed: (1) Basic research into abstract distribution systems without the constraints of a particular application, and (2) applied research into the performance and other special constraints of particular problems. Either will be much less useful without the other. (This is the opinion of the presenter, not necessarily that of the author.)

REFERENCE NO. OF PAPER: I-3

DISCUSSOR'S NAME: G. Scotti, SELENIA

AUTHOR'S NAME: J. T. Martin

COMMENT: I feel that there are several other reasons to explain why the U.K. wrote the DS0018. Can you please state the differences between 1553B and the 0018 Standard?

AUTHOR'S REPLY: The U.K. decided to produce Defence Standard 00-18 (Part 2) because 1553B was seen to be of such great use in so many applications that it was felt that it should be possible to specify the bus using a U.K. standard rather than by keep referring to a U.S. standard. The U.K. Defence Standard 00-18 (Part 2) is absolutely technically identical to MIL-STD-1553B. The differences in format and language used in Def. Stan. 00-18 (Part 2) come about purely and simply because the U.K. Authority for Defence Standards has certain rules which apply to the format and language used in a U.K. Defence Standard.

Just to reinforce and confirm:

Def. Stan. 00-18 (Part 2) is technically identical to MIL-STD-1553B. Units built to either standard will be just as compatible with units built to the other standard as if they had all been built to the same standard.

It may, however, be interesting to note that there are more things in MIL-STD-1553B, and hence in Def. Stan. 00-18 (Part 2), that are not completely specified. For instance, although some responses are defined as legal and some responses are defined as illegal there are still some responses which fall between the two definitions and the action to be undertaken in the event of receiving such a response is therefore not clearly defined.

In an attempt to promote as much standardization as possible the U.K. has, therefore, produced defined actions to be followed in the event that such a response is received. These U.K. preferred responses are documented in the guide to Def. Stan. 00-18 (Part 2). This guide has the reference Def. Stan. 00-18 (Part 1). The important difference is that whereas the requirements of Def. Stan. 00-18 (Part 2) are mandatory, the further information in Def. Stan. 00-18 (Part 1) is only advisory.

REFERENCE NO. OF PAPER: I-3

DISCUSSOR'S NAME: H. Whitehouse, USN

AUTHOR'S NAME: J. T. Martin

COMMENT: Would you comment on the properties of an avionics bus which are not provided by standard commercial buses such as the HP1B or its IEEE counterpart.

AUTHOR'S REPLY: MIL-STD-1553B has come about not just in order to redesign the wheel but because none of the commercial buses available at the time were satisfactory for the application. The requirements for a commercial interface include: high-speed capability (therefore, fast logic edges or parallel interface) and cost-effectiveness, bearing in mind the environment that the interface is to operate in. The requirements for an avionic bus include: EMC compatibility (therefore, controlled logic edges), low wiring density (to reduce weight and volume) and reliability and availability leading usually to dual bus configuration (making the use of serial transmission techniques even more important).

The above is a very brief summary of the reasons for MIL-STD-1553B. For a full account see MIL-STD-1553B Handbook or/and Defence Standard 00-18 (Part 1), the handbook and explanation for Defence Standard 00-18 (Part 2).

REFERENCE NO. OF PAPER: I-3

DISCUSSOR'S NAME: CDR J. A. Strada, ONR, London

AUTHOR'S NAME: J. T. Martin

COMMENT: How do you see the role of distributed processing in reducing crew workload and dealing with the multisensor environment in an ASW aircraft like the P3C or Nimrod?

AUTHOR'S REPLY: Distributed processing does not really effect crew workload as such. The crew should be unaware of what the design of the system that they are using is. The main item to effect crew workload is the design of the man-machine interface, this includes, of course, the keyboards, the displays and the processing which allows these keyboards and displays to function.

He said that it is true that a distributed processing system whereby the various subsystems of the system are connected together by, for instance, a 1553B bus does lend itself to the combining of information into one place and the control of a number of systems from one place. Although the same effect, as far as the crew is concerned, could be achieved without such a distributed system, I believe that you would have to pay a high hardware overhead, for instance many extra I/O control channels from the centralized system, to produce the same degree of centralization of controls and display.

REFERENCE NO. OF PAPER: I-3

DISCUSSOR'S NAME: Dr. A. A. Cai away, RAE

AUTHOR'S NAME: J. T. Martin

COMMENT: Mr. Martin has talked about the opportunities for using distributed processing in modern systems. There are many constraints which can be applied in distributing processing - such as minimum data flow, retaining comparable processing sizes, etc. In practice, however, because of the way systems are procured, and the accountability of manufacturers, does the author, as a representative of an industrial systems company, see us ever achieving anything other than functional distribution as a practical criterion?

AUTHOR'S REPLY: The problem is to fully specify the requirements to be placed on the supplier and to be able to specify the tests necessary to prove that the supplied item exhibits the attributes which are demanded. Very few manufacturers actually manufacture all items of the subsystems or system that they supply (for instance a sensor head may be purchased by a system supplier to be added into his total system or subsystem by way of a subcontract or another supplier). For these items of subcontract to be procured and accepted it must be possible to specify them and test them to that specification. If it is possible for one main or prime supplier of a system to specify such a subcontracted item, then it must be possible for some other supplier or procurement agency to also produce the necessary specification. We could therefore imagine the case where a system design is carried out by one firm to the level necessary for the equipment and subsystems specifications to be produced using as a criteria for the division of the work any split required as long as it leads to the required specifications and test specifications.

Summarizing - technically any split is possible and already achieved. Managerially, especially in the case of the procurement agencies, it may be necessary to reconsider our present working practices.

REFERENCE NO. OF PAPER: I-4

DISCUSSOR'S NAME: CDR J. A. Strada, USN, ONR-London

AUTHOR'S NAME: B. A. Zempolich

COMMENT: Reference the position of "Systems Architect" in NAVAIR. For whom would such an individual work during aircraft development? Would he stay with the aircraft as it moves into an operation status? For whom would he work then?

AUTHOR'S REPLY: (1) The PMA and his administrative division.
 (2) Yes, he/she would stay with the aircraft.
 (3) Continue to work for the PMA.

PERFORMANCE STUDY OF A DISTRIBUTED MICROPROCESSOR ARCHITECTURE
FOR USE ABOARD MILITARY AIRCRAFT

Kang G. Shin and C. M. Krishna
Electrical, Computer, and Systems Engineering Department
Rensselaer Polytechnic Institute
Troy, New York 12181

ABSTRACT

An analysis of the performance of the Distributed Microprocessor Airborne Computing System (DMACS) developed at Rensselaer Polytechnic Institute is presented. The DMACS consists of a number of quasi-independent computer subsystems loosely coupled in a highly decentralized structure that yet exhibits high cogency as a system. Some important parameters in the system such as job scheduling and starting delays, bus access delay and system reliability are studied.

In order to highlight the implications of the design options chosen, the structure of the DMACS and that of the Draper Laboratory's Fault-Tolerant Multiprocessor (FTMP) system are compared and the impact of structure on performance is discussed qualitatively.

1. INTRODUCTION

The increasing sophistication of fighter aircraft has raised the need for intelligent control equipment. All too often at present, this equipment has been added in an ad-hoc fashion. The result has been a variety of independent systems for such functions as fire control, flight control, navigation, etc. This leads to wasteful redundancy, to relatively low system reliability and a high workload upon the pilot (who is the coordinating agency). This is the main motivation for a new concept called Integrated Control (Robinson, A.C., and Hitt, E. F., December 1978; Shin, K.G., December 1979). Integrated Control (IC) is the use of control equipment as part of an organized and cogent system. Integrated Control treats the entire aircraft (the pilot included) as an organic whole. Considerable benefits follow. For one thing, equipment redundancy translates more efficiently to fault-tolerance. For another, the pilot -- while still the coordinating agency -- is no longer involved in step-by-step and detailed low level control. Instead, he is largely the decider of policy, choosing from a number of options open to him and letting the system do the rest.

Needless to say, Integrated Control requires a sophisticated computer system that is highly reliable and is capable of absorbing with equanimity the large surges of throughput demand that are characteristic of the application at hand.

A number of attempts have been made to design highly reliable systems. These include the Software Implemented Fault Tolerance (SIFT) machine of SRI International (Wensley, J.H., et al., October 1978), The Multi-Microprocessor Flight Control System (M²FCS) program of the Air Force Flight Dynamics Laboratory (AFFDL) and Honeywell (White, J.A., et al., October 1979) and the Fault Tolerant Multiprocessor (FTMP) of the Charles Stark Draper Laboratory (Hopkins, A.L., et al., October 1978). The last of these is an especially interesting design and is the result of certain well-defined design choices.

In a project recently initiated by the authors at Rensselaer Polytechnic Institute, an attempt has been made to design a high-reliability and high-throughput machine with extensive decentralization of control (Shin, K.G., and Krishna, C.M., December 1980). It has been sought to use the extended capabilities of recently developed microprocessors such as the Motorola 68000 and the AMD 2903. Delegation of control has been maximized. The system is entirely asynchronous and highly modular. Use has been made here of the essential characteristics of the application. In the first place, the aircraft mission can be rather neatly divided into nearly independent portions. This decomposition of the mission into its component parts is formalized in the concept of atom functions. Again, the nature of the application suggests a system dichotomy. This translated into the way the architecture is composed: we have a central area and a peripheral area, each with its own distinctive characteristics. The peripheral area is dedicated to particular tasks such as data-taking and actuator-driving, whereas the central area is in a symmetric formation and is therefore not dedicated to any particular task. This has obvious implications for reliability -- both the actual system reliability and the ease with which theoretical predictions concerning reliability may be made.

Also, the present architecture has been explicitly based on the concept of Integrated Control. This implies that it has been attempted to attack the aircraft control problem holistically and from a systems point of view. This is a departure from other distributed systems in that these have generally tended to consider only the computing equipment without much consideration being given to the operating environment.

This paper is organized as follows. Section 2 consists of an overview of the system architecture. This is abridged from an earlier publication (Shin, K.G., & Krishna, C.M., December 1980) and is presented here for completeness. Section 3 focuses on the central controller. The nature of the controller's functions has a decisive impact on system performance. Section 4 deals with the performance evaluation of the system. A comparison with the Draper Laboratory's FTMP is provided in Section 5 and the paper concludes with Section 6.

2. REVIEW OF DMACS ARCHITECTURE

The DMACS architecture is based on both mission decomposition and system dichotomy. The architecture has been described in some detail in (Shin, K.G., and Krishna, C.M., December 1980), but for convenience, the major aspects are briefly described below.

2.1 Mission Decomposition

The ordered set of tasks to be performed by an airborne computer system is termed a mission. A mission consists of mission segments such as takeoff, cruise, target tracking, landing, etc. Each mission segment is then divided into basic mission components called atom functions. An atom function may be considered a unit program performing a basic unit of the mission such as Kalman filtering, control law calculation, etc.

Each atom function receives raw data from its source set (of sensors, pilot-activated systems, and ground communication systems) and feeds a sink set (actuators and the cockpit display) with processed data. In view of the ever-increasing computation power of microprocessors it is not unreasonable to assume that any atom function can be handled by a single advanced microprocessor (e.g. M68000, LSI-11/23, AMD 2900 series) within the imposed time limit. This assumption together with the mission decomposition offers system modularity in both hardware and software, thereby enabling an atom function to be a hardware and software building block in the DMACS.

2.2 System Architecture

The input to the system is derived from sensors, ground communications and pilot-generated inputs. The rate of data flow is small -- only a few Hertz. The outputs of the system are to mechanical actuators and to the cockpit display. These, again, are at low data rates. In contrast, the computations themselves are generally involved and are required to be carried out at high speed.

Clearly, the processors handling the data formatting tasks from the individual sensors have to be dedicated. The processors carrying out the bulk of the processing do not have to be so dedicated.

By means of arguments similar to the above, it is possible to show that the application calls for a system dichotomy. Such a dichotomy is indeed built into the system and represented by the peripheral and central areas (Figure 1). The peripheral area consists of the sensors, actuators and associated (dedicated) processing equipment. This equipment is relatively low-capability hardware. The central area consists of high performance Processing Modules (PM's). Each PM consists of a main processor with its own private memory and two bus controllers to interface with the data and control serial bus sets. These are the only buses in the system and are triply redundant. The basic system architecture is depicted in Figure 2 and the peripheral area is shown in some detail in Figure 3.

3. MORE ON THE DMACS ARCHITECTURE

The Central Controller (CC) is the top coordinating agency after the pilot and has a decisive impact on system performance. Prior to performance analysis, therefore, it is in order to discuss the CC along with architectural implications.

3.1 Central Controller

The CC is at the heart of the DMACS and operates in two different modes; the normal and abnormal modes. The extent to which the architecture has been decentralized results in a light controller loading under normal conditions. The system can be thought of as being a set of quasi-independent computer subsystems, each member of the set being formally complete within itself under most normal conditions of operation. However, the system may behave like a centralized computer under abnormal conditions (e.g. change of mission profile).

A. Normal Mode of Operation

The central controller has, under normal operating conditions, to carry out periodic error checks and to control the allocation of the major common resource in the system -- the data bus. Data bus grant is requested and granted asynchronously according to a quasi-handshake format. The main processor within the processing module places the data word to be broadcast in the data bus controller. Bus grant requests are entirely within the domain of the two bus controllers -- insofar as the main processor in the processing module is concerned, the bus controllers represent its only contact with the outside world.

The data bus controller signals the control bus controller (CBC) that a data word is available for broadcast. The CBC responds by setting the data bus grant request bit in its transactions register. The transactions register is periodically polled by the central controller and bus grant is achieved on a first-come first-served basis.

B. Abnormal Mode of Operation

Central controller intervention on a large scale is called for when abnormal events occur. These may call for a redistribution and/or redefinition of system resources. The following are the most commonly encountered abnormal occurrences:

- Malfunctions in PM's
- Mission profile changes
- Test requests from the peripheral area.

To handle these occurrences, the CC needs precise, accurate and timely information on the status and duty of each processor in the system. The principal table of information held within the central controller is the Central Cluster Status Table (CCST). The CCST has the following format:

ATOM FUNCTION	ACTIVE/ PASSIVE	PROCESSORS ASSIGNED	PROCESSOR STATUS

The atom functions are ordered according to their importance. Processors not assigned to any atom functions (i.e. free PM's) are listed as being assigned to atom function 0 (i.e. the lowest priority atom function). Processor Modules that are malfunctioning are assigned an atom function number one higher than the most critical function of all -- the control function.

The active/passive column indicates whether or not the concerned atom function is active within the current mission profile. (Note that all atom functions possible are listed: not just those that are currently active. This does not cause a time penalty for table-search during reallocation due to the way the table is structured.) Inactive functions generally do not have any PM's assigned to them.

When a PM malfunction is reported, the central controller scans the CCST from the bottom. If -- as is generally the case -- there is a free PM available, that PM is brought into the depleted triad.

In the event that there is no free PM available, the least critical atom function is retired and the PM's assigned to it are used as spares.

The process of triad reconfiguration is as follows:

1. Delink the injured processor.
2. Find a replacement PM.
3. Load status.
4. Check status and induct into the system.

Of these steps, steps 1, 2 and 4 are controller-intensive, i.e. they require extensive controller involvement. Step 3 on the other hand is handled without much reference to the controller. Transfer of software is carried out by DMA.

A slightly more complicated process is involved when triad reconfiguration is called for. The loading upon the controller is far greater than in the case of a random processor knockout (i.e. the random demise of a PM). Also, the volume of software to be transferred is far greater. The latter reason is the more straightforward to handle: the time required on a bus for software transfer is very nearly proportional to the volume of software, while the controller loading is more difficult to quantify precisely.

The effect of controller loading upon the system is minimized by carrying out the reconfiguration in stages, configuring the most important triads first so that the more critical new functions are activated as soon as possible. Note that functions such as flight control are active throughout and are not affected by reconfiguration except to handle malfunctions.

3.2 Implications of the Architecture

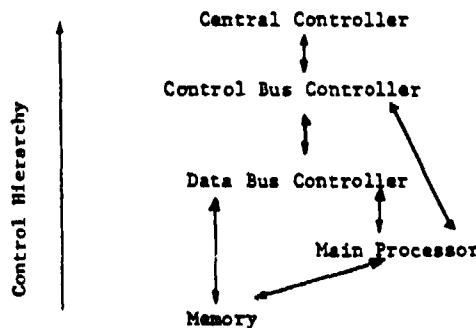
There are some particular aspects of the application we are concerned with and some points in the architecture here presented that are worth further discussion.

The most important point to consider in aircraft control is that the atom functions into which the mission divides are essentially decoupled. Flight control, fire control and navigation (to name just a few atom functions) affect different actuators. The application at hand is characterized by the fact that while different atom functions might be triggered by common sensor inputs, the sink sets of the individual atom functions are generally distinct. It should be noted here that the pilot is still the overall coordinating agency -- albeit at a much higher level than in the conventional method.

From this fact follows the present architecture which is not so much a true multiprocessor architecture (Enslow, P.H., March 1977) as it is a collection of cooperatively coupled computer systems that require controller intervention at a low level for most of the time.

A second point worth considering is the existence of two distinct bus sets -- the data bus set and the control bus set. The control bus simplifies the executive software considerably and reduces the need for tight synchronization in the system.

Linked with the idea of a control bus permanently captured by the central controller are the bus controllers and the architecture of each PM. The PM admits of considerable internal decentralization. The two bus controllers -- which are actually dedicated processors with their own buffers -- handle transactions with the outside world. The Control Bus Controller (CBC) is the "local arm" of the central controller. The CBC is activated by central controller command and is thus entirely under central control; but it has sufficient intelligence to reduce controller loading. (An analogy may here be drawn between the above and the channel and device controllers in a commercial computer system. With a modestly intelligent device controller, the channel controller simply needs to initiate device action and let the device do the rest until a device end is encountered. Examples of device controllers are disk controllers, card-reader controllers, etc.) The control hierarchy in DMACS is as follows:



4. SYSTEM PERFORMANCE

4.1 Job Starting and Job Scheduling Delays

Job scheduling delay is defined as the duration between a job request and the allocation of the system resources for the execution of the job. Job starting delay is the time delay between the job request and the actual execution of the job.

Due to the quasi-static scheduling policy followed, job scheduling and job starting delay are relevant only when jobs need to be scheduled; i.e. at moments of change in mission profile or during PM replacements.

Changes in mission profile are brought about when there is a demand for a new set of atom functions. The varying nature of mission requirements suggests two choices: either allot a separate PM triad for every atom function (whether required in the current mission profile or not) or allot PM triads only on demand. The second approach is the more practical and is adopted here. There are certain flight functions such as flight control and navigation that must remain operational throughout the mission lifetime. Others such as those used for landing and takeoff are available on demand. It is the scheduling delay for these jobs that we are primarily concerned with: job scheduling delays do not ordinarily affect the "perennial" atom functions such as those cited above.

Job starting and scheduling delays are expressed as follows:

$$\text{Job starting delay} = t_1 + t_2 + t_3 + t_4$$

$$\text{Job scheduling delay} = t_1 + t_2$$

where

t_1 = time taken by the controller to take action upon the request for that particular atom function.

t_2 = processor allotment time

t_3 = software transfer time

t_4 = processor check time

Of these times t_1 and t_3 are highly variable; t_2 and t_4 are not so inconstant.

Time invested by the central controller in reconfiguring the atom function is the sum of t_2 and t_4 , which is relatively small and constant. The rate determining step in job starting delay is either t_1 or t_3 depending on central controller loading. Except under the most difficult conditions, $t_1 < t_3$ which indicates t_3 as the rate determining step. t_3 is the ratio of the volume of software transferred to bus bandwidth.

As estimation of the values of these variables is not easy. However, an order-of-magnitude calculation may be attempted as follows.

Processor allotment consists of two stages: 1) Find a PM that is available
2) Update the CBC Table and the CCST.

Step 1 involves (a) accessing of a record from the CCST, (b) checking its suitability, and (c) deciding whether or not to terminate the search.

For a moderately fast system, accessing a record should take much less than 1 μsec . Checking its suitability involves computing a Boolean function that, again, should take somewhat less than 5 μsec (we assume a clock rate of 10 MHz). The step (c) is essentially an appendage of (b).

It follows therefore that the total time (in μsec) taken in locating a suitable PM is less than six times the number of accesses (typically 1). It is usually less than 12 μsec even under poor conditions.

Once a PM has been located, updating the tables entails entering some four variable values in the CCST and the atom function number in the CBC transactions register. This should involve less than 10 clock cycles per entry making 50 clock periods in all, or about 5 μsec .

Each applications program has a bootstrap portion that loads into the CBC table the variables of interest. These are the variables the bus controllers are to recognize as being relevant to the particular atom function in hand. This does not usually take more than 100 μ sec. Hence $t_2 \leq 100 \mu$ sec, and t_4 or the status check time is very small and constant. The PM in question runs a test program and sends the results to the controller. The controller has only to match the answers with the right ones (held in its private memory) to determine processor status.

It follows then that the total central controller time invested per PM reconfiguration approximates 100 μ sec. Hence t_1 is now estimable by:

$$t_1 \approx \text{housekeeping time for normal activities} + t_2 * n$$

where

$$n = \# \text{ of PM's configured after request was received from the concerned PM.}$$

The allotment of PM's proceeds on a priority basis. The controller scans the atom functions that are to be represented by the PM's and chooses the most important or critical atom function from amongst these for implementation. This procedure ensures swift implementation of the more important atom functions.

The job scheduling and starting delay for reconfiguring individual PM's ideally have identical profiles -- only the constants involved are different.

One possible outcome of the PM induction procedure is that under extreme circumstances it may so happen that the least important atom function will never get assigned. This could be forestalled by automatically updating priority as a monotonically increasing function of waiting time. We choose, however, not to do so since the more critical functions must never be impaired for more than the minimum possible duration. In any case, as we shall see, this problem is more academic than real.

Figure 4 depicts the job scheduling delay as a function of the precedence in the job request queue. The precedence in the job request queue is easy to determine. We have two distinct conditions under which allocation is carried out. The mission profile may change or processors could suffer random knockouts. The former case involves an entirely new set of atom functions simultaneously being required. The precedence in the waiting queue is then simply the relative importance of the function in relation to the others in the new set.

A more complicated case (theoretically speaking) arises in random knockouts. As was mentioned earlier, it is entirely possible that the random knockout of processors should occur at such a pace and in such a sequence as to effectively kill a lowly atom function. This can, however, occur only when more than one failure occurs more or less simultaneously. This is highly improbable. The probability of failure of a PM is around 10^{-4} per hour. Reconfiguration takes less than 100 μ sec for the controller to achieve. Probability of a processor failing in that time is approximately 10^{-11} . For any atom function to be kept waiting for central controller attention for time T, the number of more critical PM's that must fail is $T/100$ since 100 μ sec is approximately the time required by the central controller to reconstitute the injured triad.

4.2 Bus Access Delay

The system consists of a set of processors communicating by means of two sets of buses -- a data bus and a control bus; both triply redundant for adequate fault tolerance. The control bus is permanently captured by the central cluster controller and employed in such activities as test initiation, bus grant and rebroadcast command as well as the DISCONNECT command issued by the central controller to a failed processor. The data bus is allocated to whichever processor needs it on a First Come First Served basis. The average access delay and maximum access delay as a function of the bus demand profile are studied.

The actual procedure for determining delay is very simple. Bus grant requests are put into a time indexed array (a list) in the order in which they arrive. The central controller steps through the items in the list granting access to the oldest item still outstanding. The time at which this bus grant is achieved is noted and the delay is computed by subtracting the request arrival time from the bus grant time. Using these figures, it is possible to arrive at values for the maximum wait time for bus grant and for the average wait time. Both parameters are of interest in evaluating the system; they have an important role to play in the validation process.

The specific case we have described here is for 20 central cluster requests per "request cycle". The figure of 20 may appear somewhat arbitrary, but in fact represents a system of typical complexity. Again, we're looking more for the shape of the response profile than for actual numerical values.

The input arrival rate profiles studied are as in Figure 5. They therefore range from the uniform (1 request per interval) to the very bursty (20 requests in the first frame; 0 elsewhere). The average and maximum delay values that results are graphed in Figure 5.

4.3 System Reliability*

Reliability is a measure of the probability of failure. In a system as complicated as ours, there are clearly many classes of failure. These are listed below.

Type 1 failure: A type 1 failure is said to have occurred when the capacity of the system to compute a particular atom function has been permanently removed. (By 'permanently' we mean of course till the system is manually serviced). Since there are many atom functions, more than one type 1 failure can have occurred in the system at any one time.

* This portion is drawn from (Shin, K.G., and Krishna, C.M., December 1980).

Type 2 failure: A type 2 failure is said to have occurred when the capacity of the system to compute a particular atom function has been temporarily removed. We subdivide this class into two subclasses.

Type 2a: A type 2a failure occurs when the impairment of system function has occurred for the time needed to switch from active to backup units. This time is relatively small.

Type 2b: A type 2b failure occurs when the impairment lasts for as long as it takes to reallocate functions among the central cluster processors.

Generally, a type 2b failure takes much longer to recover from than does a type 2a failure.

Probabilities of failure can be deduced as follows:

Let

m_i = number of output actuator triads forming the sink set of atom function i .

n_i = corresponding number for sensor triads in source set of atom function i ,

p_{sens} = probability of a sensor failure.

p_{act} = probability of an actuator failure.

p_{bus} = probability of a bus failure.

p_{proc} = probability of processor failure.

It bears pointing out at this stage that the above probabilities are very small; we note that a typical range is 10^{-4} to 10^{-7} per hour. With these values in mind;

Probability of a type 1 failure

$$P_1 = \sum_i [n_i p_{sens}^3 + m_i p_{act}^3 + (m_i + n_i) p_{proc}^3] + 2p_{bus}^3$$

Probability of a type 2a failure,

$$P_{2a} = \sum_i [m_i p_{act} + (m_i + n_i) p_{proc}] + p_{proc} + 2p_{bus}$$

Probability of a type 2b failure,

$$P_{2b} = 3\alpha p_{proc}^2$$

where α = number of atom functions in the mission. To obtain a feeling for the actual figures involved, we employ the following probability estimates:

$$p_{proc} = 10^{-4}/\text{hr}, \quad p_{act} = 10^{-6}/\text{hr}, \quad p_{sens} = 10^{-6}/\text{hr}, \quad p_{bus} = 10^{-5}/\text{hr}.$$

Assume simply that all atom functions are identical with respect to hardware requirements and

$$n_i = 2 \quad \text{for all } i, \quad m_i = 1 \quad \text{for all } i, \quad \alpha = 15.$$

In such a case,

$$P_1 \approx 0.5 \times 10^{-11} \text{ per hour}, \quad P_{2a} \approx 0.5 \times 10^{-3} \text{ per hour}, \quad P_{2b} \approx 10^{-6} \text{ per hour}.$$

Note here that a type 1 failure is the only true failure in the system sense; type 2a and 2b failures result in system reconfiguration with some loss of throughput, but no system impairment of more than a temporary nature.

5. COMPARISON WITH FTMP

We turn now to comparing two similar architectures: the DMACS and the C. S. Draper Laboratory's Fault-Tolerant Multiprocessor (FTMP). It is not our intention in this section to seek to make definitive judgments upon the relative worth of the systems -- only to describe the implications of a set of design options taken in each case.

The FTMP is, in hardware terms, superficially similar to our architecture. For instance, it is a bus-oriented machine, with triple redundancy being used throughout for the detection and correction of errors.

The Draper Laboratory has essentially chosen a different set of options from ourselves. A study of the differences together with a brief overview of the implications is worthwhile since it brings out in sharp relief the tradeoff options available to the systems architect.

The major points of difference are:

- The processors in a triad operate in tight synchronism in the FTMP while operation is completely asynchronous in the DMACS architecture.
- FTMP is essentially the central portion of an aircraft computer control facility; data acquisition and delivery are not considered in much detail. The DMACS architecture explicitly incorporates sensors, actuators and associated processing equipment into the system.
- Job scheduling in FTMP is completely dynamic; the system controller is, for all practical purposes, a job scheduler. The DMACS system involves quasi-static job scheduling.
- The bus structures are different. FTMP has a "Mass Memory Bus", an "Internal I/O Bus" and an "External I/O Bus" while the DMACS makes do with just two sets of buses: a data bus set and a control bus set.
- The basic processing module is far simpler in FTMP than in the DMACS.

We provide below a more detailed exposition of the consequences of the differences noted above. In the FTMP, all elements of the multiprocessor operate using a common time reference. Four mutually phase-locked clock generator modules operating together provide a fault-tolerant time reference (Lala, J.H., & Smith, C.J., 1979). The effect of this tight synchronism is to make data transfer between processors and memory and processors and processors simpler and therefore faster. A common clock obviates the need for a pseudo-handshake as is used in the DMACS architecture. However, for this benefit in lowered intercommunication complexity, we have to pay in terms of reduced reliability. The disabling of the clock will be disastrous to the system; and while the existence of four clock modules phase-locked to each other assures considerable fault-tolerance, the synchronism nonetheless introduces an additional potential source of catastrophic failure. An additional consequence of this is seen in the consideration of the third point in the above to which we shall come.

FTMP has an External I/O bus and an External I/O port that handle data input and output. No restriction is therefore placed on the hardware acquiring and using data: the configuration of the sensors and actuators is undefined. This makes for easier adaptability to existing systems. The FTMP can therefore be "added on", so to speak.

On the other hand, the DMACS architecture imposes a certain structure upon the actuators and sensors. The reason is that we felt that characterizations of the system would be invalid if they did not include the communication with the environment as part of the systems -- and this, is after all, the very reason for the existence of the computer system in the first place.

Job scheduling in the FTMP is entirely dynamic. This is justified by the Draper Laboratory after consideration of the alternative which is the synchronous job scheduler. In synchronous job scheduling, periodic jobs are completely prescheduled with each job occupying a certain predefined time slot in the schedule. The main advantage is low central control overhead. It is claimed by the Draper Laboratory that the major disadvantage of this kind of algorithm is the lack of flexibility and the complexity of preassigning jobs to processors in a three-unit multiprocessor. Again, failure of one of the processors in a triad or changes in job parameters such as iteration rates, may require a totally new schedule. The synchronous scheduler is therefore not adopted in FTMP (Lala, J.H., and Smith, C. J., 1979). Instead, an entire scheduling is carried out whenever an atom function has to be executed.

The problems pointed out in the remarks above are very real; but we believe they follow partly from the tight synchronism the FTMP is forced to operate in. In an asynchronous and highly decentralized system -- such as ours -- all the advantages of on-time job execution with practically no delays and a high load factor are available (as we have seen in the performance characteristics) without the disadvantages mentioned above.

Again, when the mission profile changes, requiring a large-scale reallocation, the applications software for the new atom functions thereby introduced are loaded (in the DMACS) using DMA and a conceptually simple procedure. Reconfiguration time in such cases is very low.

Our bus system is conceptually simple. All data (whatever its origin) is treated in the same way and broadcast on the data bus according to the same format. This simplifies malfunction detection and handling and makes the systems software less complex. A control bus is used in addition to the data bus to simplify central controller intervention in the system. The bus structure of FTMP is much more complex. While it does not necessarily follow that a reduced reliability is the result of such increased complexity, it is, in our opinion, to be avoided wherever possible.

All differences in structure and performance between the FTMP and DMACS can be held to issue from one basic difference in design philosophy: FTMP IS LESS DECENTRALIZED THAN OUR SYSTEM. The central controller has a major role to play in finding a processor every time an atom function is to be executed; no matter whether the atom function is periodically required or not. The central controller -- which as has been pointed out is basically a job scheduler -- is thus involved in scheduling even continuously periodic functions. The result is a continuous high loading upon the controller and a relatively high overhead in the form of applications software transfer. This may result in needlessly slowing down the system.

The DMACS architecture, on the other hand, follows a consciously laid down policy of maximum decentralization. The central controller is involved in regular activities mainly in arbitrating access to the data bus. Regular housekeeping chores are therefore not time-consuming. This has the merit that when an abnormal

event takes place the controller response is faster than it would otherwise be. The controller delegates many of the routine chores to the control bus controllers in the various processing modules.

It is worth pointing out that the FTMP system is far older than our own. Consequently, it has undergone more detailed analysis than the DMACS. For one thing, a prototypical version of FTMP has been constructed while our system is as yet in the design stage. All our remarks should therefore be read in this context.

6. CONCLUSIONS AND DISCUSSION

This paper has sought to describe a distributed processor structure for the effective control of military aircraft. The goal has been to configure, out of components of military-grade reliability and easy availability, a computer system that is at the same time easy to expand, service, program and that is reliable and flexible enough to accept a considerable number of alterations without requiring a major revision of the basic structure.

This project was motivated by a desire to employ the concept of Integrated Control in fighter aircraft. Ad-hoc addition of components to aircraft results in wasteful redundancy that does not necessarily translate into increased real redundancy from the performance point of view. Again, there is the very real possibility of one element in the system affecting the performance of another; thus degrading the overall system performance. This is clearly an unsatisfactory state of affairs but one that occurs frequently in extremely complex systems. The conception and design of the system as a whole generate certain problems. However, the holistic design of complex systems provides one with an opportunity to carry out optimization with respect to the whole system and not just with respect to one isolated component portion of it. By pooling all resources into one large system it is possible always to provide increased reliability to the more critical functions. Fighter aircraft today are designed to fly to the edge of instability and designers push the inherent properties of the basic mechanical structure of the aircraft to the maximum possible extent. In such a dynamic -- and not always friendly -- environment, it is essential that the reliability of the basic critical flight functions be extremely high. The high reliability required of any system used aboard a manned aircraft has to be achieved by using components that by themselves are far less reliable than that. Commonly used figures for the reliability of processors peg the reliability at around 10^{-4} failures per hour. Mechanical devices such as actuators do not show a very great improvement upon this figure. It is therefore contingent upon the structure or the configuration of the computer system to create, out of components that are by themselves not very reliable, a super-reliable system.

The requirement of high throughput is no less important than that of reliability. The fighter aircraft operates in a highly dynamic environment and much of the data from the sensors has to be processed in real-time. The environment is characterized by rapid surges in demand upon the services of the computer system. The system must therefore be robust enough to absorb such surges without a lowering of reliability.

A third requirement is ease of programming and system flexibility. A system that is not easy to program is potentially very expensive to operate and is prone to errors. System flexibility and modularity are required for ease of servicing and maintenance.

The present system is based upon the three basic requirements listed above. Reliability is provided through the use of triple-modular redundancy with voting and a conceptually simple structure.

A high throughput (or low bus-access delay which is equivalent to high throughput in our case) is achieved by means of using two sets of buses instead of just one. The control bus triad serves essentially two purposes: first, it lowers the demand upon the data bus and second, it provides the central controller with a simple means of propagating control signals. Controller intervention into system activity is not delayed by ongoing transmissions upon the data bus.

The modularity that is built into the system provides ease of programming together with expandability and improved serviceability.

It is clear, therefore, that the configuration arrived at is a direct result of the requirements of reliability, ability, high throughput and flexibility.

There are, however, many interesting problems not yet studied in any great depth. The simulation of the present structure has been partial and with respect only to specific characteristics such as job scheduling delays, reliability and bus access delay. A more complete simulation package for the system is planned.

REFERENCES

- [1] Enslow, P. H., March 1977, "Multiprocessor Organization - A Survey", Computing Surveys, Vol. 9, No. 1, pp. 103-129.
- [2] Hopkins, A. L. Et al., October 1978, "FTMP - A Highly Reliable Fault-Tolerant Computer for Aircraft Control", Proceedings of the IEEE, Vol. 66, No. 10, pp. 1221-1239.
- [3] Lala, J. H. and Smith, C. J., 1979, "Performance and Economy of a Fault-Tolerant Multiprocessor", New York, Proceedings of National Computer Conference, pp. 481-492.
- [4] Robinson, A.C., Hitt, E.F., December 1978, "Integrated Control - A Unified Approach to Management of an Aircraft", Task Final Report, AFFDL/AC, Contract No. F33615-76-C3145.
- [5] Shin, K. G., December 1979, "System Architectures for Implementing Integrated Control Approach to Management of a Military Aircraft", Final Report, AFFDL/AGC, Contract No. F 33615-76-C3145, Request No. 49.
- [6] Shin, K.G., and Krishna, C.M., December 1980, "A Distributed Microprocessor System for Controlling and Managing Military Aircraft", Miami, Florida, Proceedings of Distributed Data Acquisition, Computing

and Control Symposium.

- [7] Wensley, J.H., et al., October 1978, "SIFT, The Design and Analysis of Fault-Tolerant Computer for Aircraft Control", Proc. IEEE, Vol. 66 No. 10, pp. 1240-1255.
- [8] White, J. A., et al., October 1979, "Multi-microprocessor Flight Control System Architectural Concepts", Los Angeles, CA., Proceedings of AIAA Computers in Aerospace Conference, pp. 87-92.

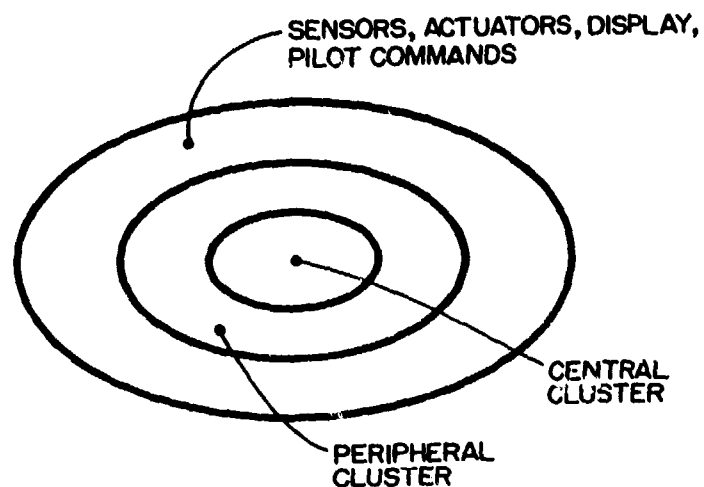


Figure 1. Overview of System Architecture

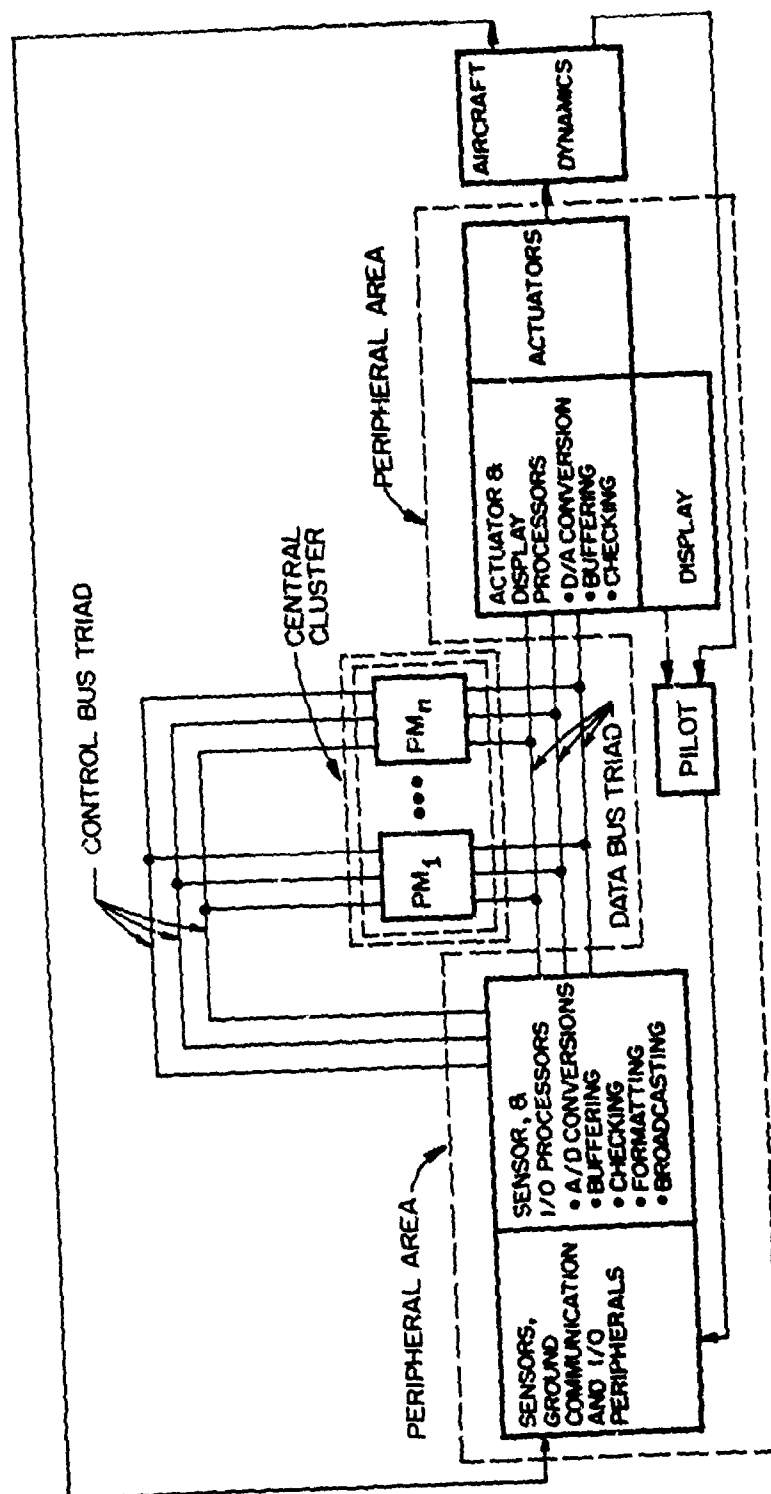
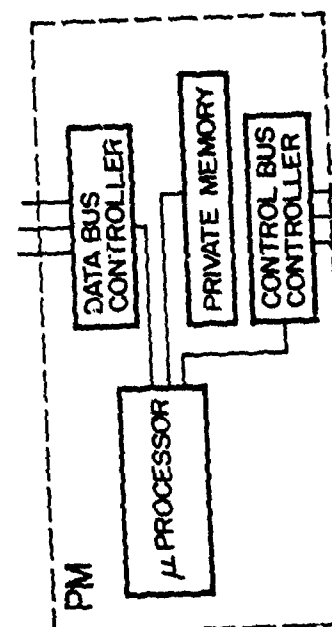


Figure 2. Architecture Detail



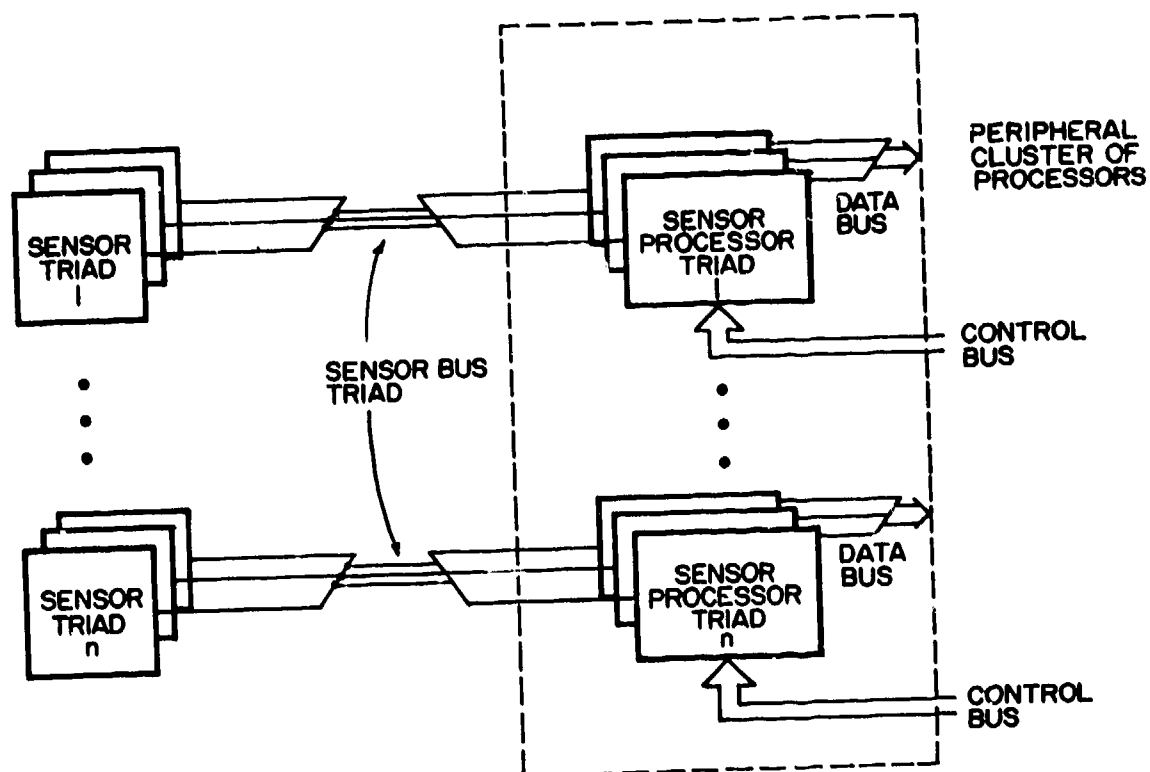


Figure 3a. Peripheral Area Detail

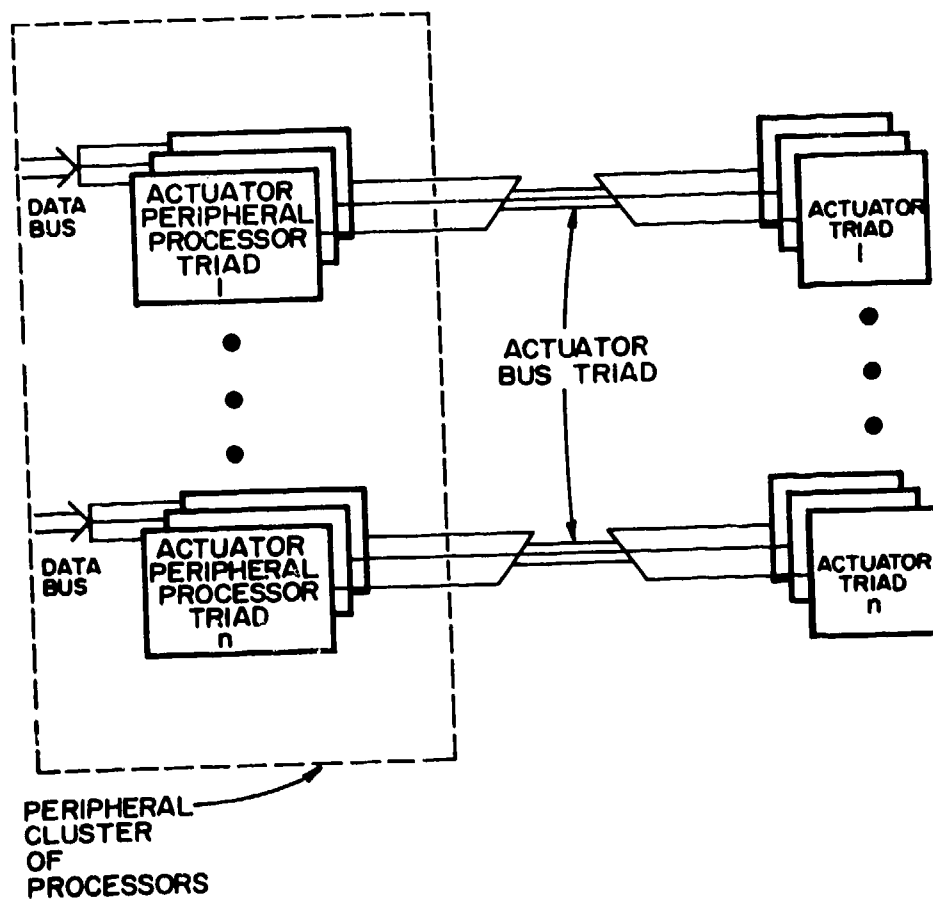


Figure 3b. Peripheral Area Detail

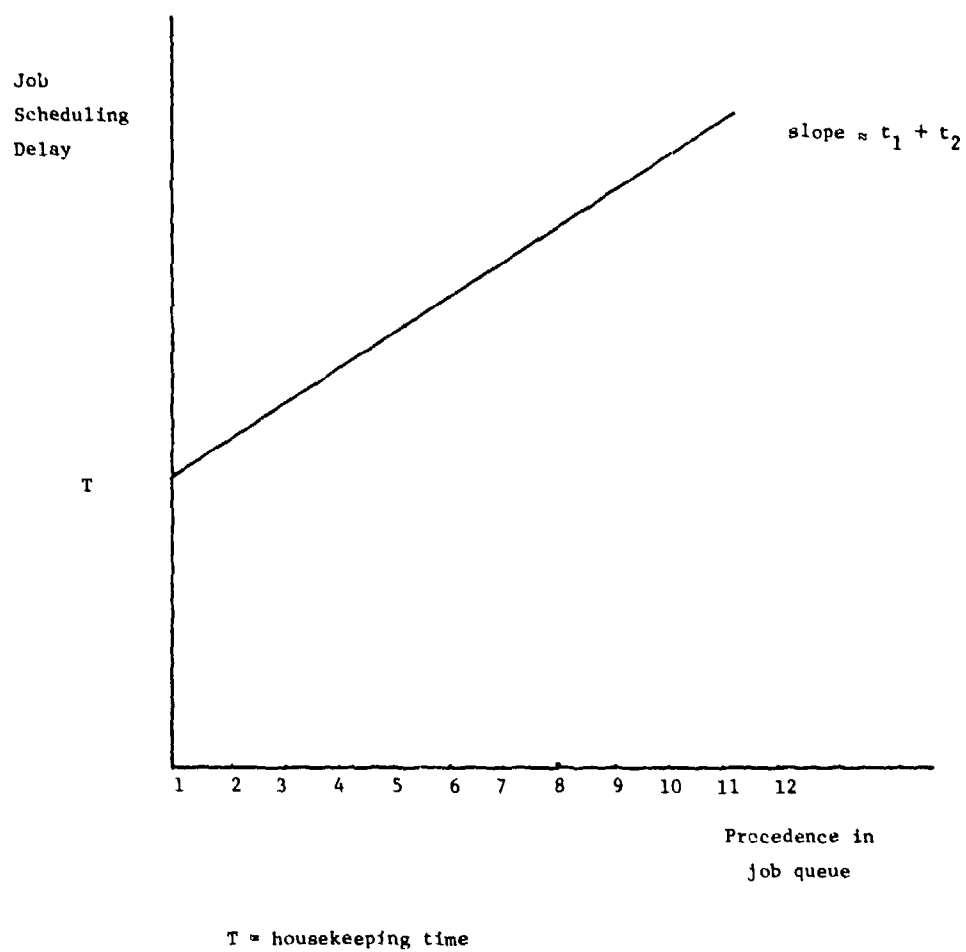


Figure 4. Job Scheduling Delay

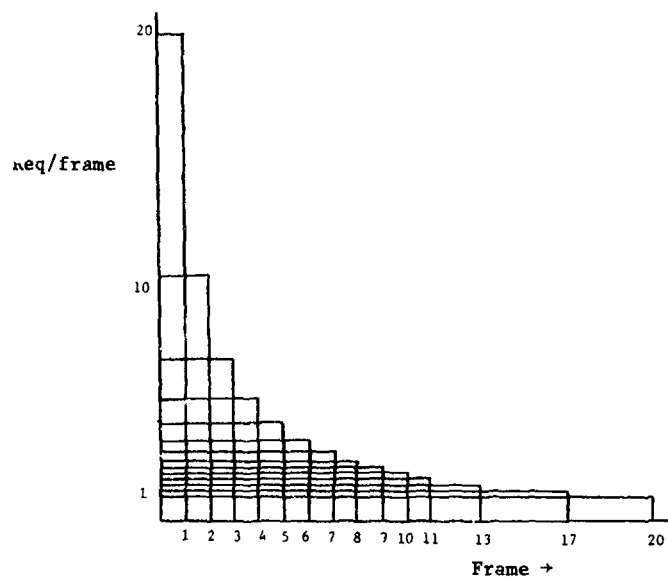


Figure 5a. Bus Request Profile

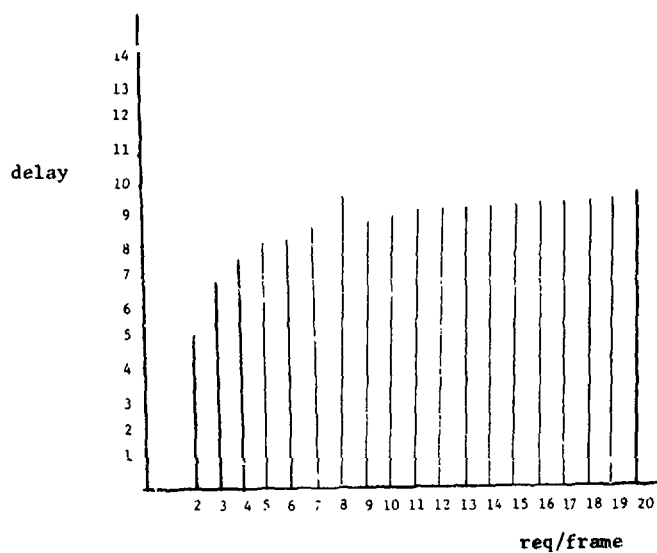


Figure 5b. Average Bus Access Delay

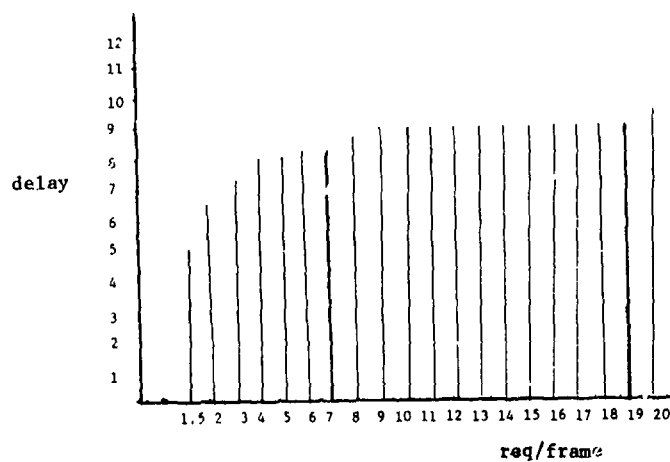


Figure 5c. Maximum Bus Access Delay

THE DEVELOPMENT OF ASYNCHRONOUS MULTIPROCESSOR CONCEPTS FOR
FLIGHT CONTROL SYSTEM APPLICATIONS

S. M. Wright and J. G. Brown
British Aerospace
Brough
North Humberside
United Kingdom

SUMMARY

In early 1979 a limited research investigation was initiated to examine the possible impact that recent advances in large scale integrated circuit technology might have if applied to flight control systems. The initial studies concentrated on alternative digital processor architectures.

One promising research avenue was identified as being the use of multiple microprocessors, each functionally dedicated and running asynchronously with a short program cycle time. This approach promises benefits in a number of areas:

1. ease of generating/proving high integrity software
2. reduced propagation delays
3. reduced hardware/software synchronisation overheads
4. retention of classical feedback control design techniques
5. extendable processing power

Part of the ongoing Active Control Technology activity at BAe Brough is an involvement in a flight dynamics research programme using a RAE Hunter aircraft converted to fly-by-wire. This programme has identified a need for a flexible digital flight control processor for such research and has provided a focus and stimulus for multiprocessor studies. As a result BAe Brough are now in the process of developing a multiplex digital full authority flight control computer for this specific application, with a view to installation in the aircraft.

Because of the short timescale of this particular application certain questions which would relate to a production system have been circumvented rather than resolved. However, this does not affect the concept which is considered to be of considerable interest and relevance to future systems.

1. INTRODUCTION

As aircraft designers have striven for more and more aerodynamic performance, the aircraft's natural stability and control characteristics have deteriorated. This has required the application of increasingly complex feedback control systems in order to artificially restore good handling characteristics. This process has reached the stage where some current and most projected combat aircraft are totally reliant on the correct and continuous operation of these active control systems. Typically, present generation aircraft use analogue computation which is multiple redundant in order to achieve the integrity targets necessary for a flight critical function e.g. Tornado or F16.

Analogue computation, however, restricts the type and complexity of control law which can be applied. It can also cause production and/or maintenance problems in achieving the required level of matching of characteristics between one computer and the next, and it can be difficult to modify the control characteristics.

These problems have led designers towards the application of digital processors to computation of the control action since they promise to substantially reduce all the above problems. Digital computers offer the additional advantages of being able to incorporate extensive built-in and pre-flight tests, together with a reduced size and weight. The overall architecture of the system has remained as a number of identical lanes each designed around a central digital processor. In practice it is becoming increasingly apparent that the software costs associated with such systems are very high. These costs are principally associated with the flight critical nature of the computation since if there were any faults in the software, then it would occur simultaneously in all lanes of the system causing possible loss of the aircraft. In order to minimise the possibility of such a failure, great reliance is placed on independent and comprehensive cross checking of the operation of the control program, and on the management system established to ensure compliance with these safeguards. Further costs are introduced by the need to produce large amounts of code which must be optimised so that any computation time delays are minimised. This involves programming at assembler level, and this in turn requires the establishment of a dedicated team of experienced programmers. This results not only in high costs but also in long timescales from control law specification to the production of verified software. This may be acceptable for a production aircraft but certainly not for a research aircraft, and probably not for the development phase of a new aircraft since here the ability to rapidly modify the control characteristics is essential.

Even after the most comprehensive software testing there will still be some concern that there could be some latent fault present in the program which would only manifest itself under a particular combination of circumstances, resulting in a catastrophic failure. This is due to the very large number of states that a digital processor can enter, corresponding to different data values and paths through the program. Because of these problems it was considered worthwhile re-examining the basic concept of the central digital processor to see if a different hardware approach could ease the task of software generation, particularly with reference to research and development aircraft but potentially for general application. The aim of this investigation was to reduce the magnitude of the software task to a level where it could be accommodated by an on-site team, to suggest ways of generating visible testable software less prone to context dependent failures and to provide a system with the type of excess computational power that would allow the convenient investigation of advanced control concepts. This approach was considered viable firstly because the amount of programming normally associated with control functions could be less than 25%

the rest being accounted for by various housekeeping functions such as consolidation, built-in and pre-flight test, failure management etc. (Ref. Corney 1979). Secondly, because of the considerable advances being made in the field of large scale integrated circuit technology, particularly in the area of microprocessors.

2. PROPOSED PROCESSOR ARCHITECTURE

2.1. Relevance of a multiprocessor approach

The first step is to attempt to partition the software into small modules which can only interact in a limited number of well defined ways which, if possible, should ease the specification, modification and testing of the program. This type of partitioning has been implemented on ground based systems using specialised operating systems e.g. MASCOT (ref. Jackson K. and Simpson H. R. 1974) and the considerable advantage that the interface between modules is so well defined that an individual module can be removed, modified and replaced without requiring re-validation of all the other software modules comprising the system. However, this technique implies the use of complex executive software which in itself would be difficult to develop and validate to the required level of confidence, and would be an extra overhead on the control processor's time. Hence both the software and hardware need to be partitioned, i.e. each software module can be allocated its own dedicated processor such that all the tasks run in parallel.

Since the constituent processors in such a system are running in parallel then the computation time delay can be significantly reduced, hopefully to a value where it becomes insignificant. This should relax the otherwise stringent requirement of producing highly optimised code to minimise execution time. It could also simplify the control law design task since, if time delays are insignificant, then the analysis and design of the control system can be accomplished using classical linear control theory with no need to resort to Z transform techniques.

An additional advantage of a multiprocessor configuration is that it provides extendable processing power, thus allowing flexible incremental enhancement of the control capabilities of a system, either in response to new applications, or to a gradual development of the original application.

2.2. Choice of a communication strategy

The traditional difficulty with a multiprocessor system lies in the choice of a communication strategy. If a bus structure is chosen then the throughput of the bus can constrain system expansion or introduce variable time delays dependent on bus loading. If a multipoint memory technique is used then this limits the number of processors which can be attached. More advanced concepts such as packet switching networks, could comprise a research programme in their own right. With any of these systems it is difficult to constrain access between processors such that the effect of a software fault in one processor cannot cause unpredictable software faults in other modules.

A network communication strategy does not inhibit system expansion since the number of links can be increased indefinitely to accommodate the extra data traffic caused by additional processors. If the links are constrained arbitrarily to carry data but not control information then the effects of failure in any one module become predictable, hence allowing the possible containment of failures which occur in non flight critical sections of code. This should either improve reliability in operation, or reduce the burden of testing. This inherently rigorous control over the interfaces between software modules also reduces the potential for adverse interaction between the sections of code produced by different members of a programming team either during the initial program development phase, or after modification of an individual module, since the structured programming concept of having locally defined variables only available locally is implicit in this system.

Typically the data being transmitted between processing modules in a real time control system consists of a number of variables each representing a continuous function of time. If each variable is allocated a unique communication channel then the operation of the system can be conveniently monitored, hence easing testing and acceptance procedures. Since each variable represents a continuous function of time and cannot be overwritten except by a more recent value, then it becomes possible to dispense with the need for handshake routines, interrupt handlers, or other software protocols, again easing the programming task.

Thus we have the concept of an asynchronous multiprocessor flight control computer. If this computer can be made from a number of identical hardware blocks then there is also potential for reduced hardware costs and increased flexibility in configuring a control computer to meet different applications, requiring perhaps different levels of redundancy, or processor power. This is in addition to the system design, programming, and integrity benefits already suggested.

3. EXPERIMENTAL MULTIPROCESSOR SYSTEM

3.1. Choice of Processor

Having decided to investigate the implications of this type of asynchronous multiprocessor concept, the first task was to choose a commercially available microprocessor which could demonstrate the major features of the idea without involving an extensive hardware development programme. The relatively large number of processor modules anticipated for a practical system focused our attention on single chip microprocessors in order to keep the volume of the final system within practical bounds. One simple method of achieving asynchronous communication between processors is to use analogue interconnections; this, together with a requirement for high processor speed and UV Erasable PROM programming, narrowed the available field down to one device, the Intel 2920 Analogue Signal Processor (See figure 1 for a functional block diagram and functional specification of this processor.)

The choice of a device with analogue interfaces also allows convenient integration with existing analogue flight control systems with which we are involved and offers the possibility of enhancing these systems and developing control computing ideas in parallel.

There are, however, some characteristics of this device which, while they would not affect any concept proving exercises, might prejudice application to a realistic control task. Viz:

1. I/O resolution of only 9 bits (internal resolution of 25 bits)
2. Limited instruction set (no branch instruction)
3. Limited program space
4. New device of uncertain detailed characteristics

It was decided to proceed with a concept proving exercise since it was anticipated that, if successful, then the above limitations could be overcome if necessary on a fully engineered system, probably using a different processor. Alternatively this initial study might show that the short term expedient application of this device to current flight control tasks was practical.

The resolution of the analogue input and output is at least two bits less than required but, since the relevant interfaces are an integral part of the device, it was impossible to alter the hardware characteristics. A software technique was therefore developed which (at the expense of a small external hardware modification) enabled bandwidth to be traded for improved resolution. This technique has been shown to be capable of achieving a three bit enhancement with a reduction in interface bandwidth from 8 kHz to 1 kHz. See Fig. 2 (Ref. Wright and Fletcher 1980).

The lack of branch instructions is an advantage from the point of view of software testing since it dramatically reduces the number of possible context dependent failures. It is also of benefit in the coding of linear control functions since every instruction is executed every program pass and hence the iteration rate is constant, independent of the software (equal to 0.12 ms for the 2920-16 operating at a 600 ns cycle time). However, combined with the limited instruction set and program size, this cast doubts on the ability to use the device to implement complex control functions, particularly those involving logarithmic and trigonometric functions, and so several exercises were undertaken to test this.

In one case a waveform generation/co-ordinate transformation task was programmed on one processor, including full four quadrant sine/cosine functions (See Fig. 3 and Ref. Wright, 1980). In a second exercise the programming of aircraft control laws was investigated. This study indicated that a typical longitudinal control system, including gain scheduling and special high incidence control laws, could be implemented on four processors (Ref. Sharaz, 1980). Figure 4 illustrates a representative flight control law element.

The detailed hardware characteristics of the device are still undergoing extensive testing but it appears that if suitable precautions are taken then this device can be applied in the short term to a number of control tasks.

3.2. Standard computing and interface modules

In order to use this device in a range of applications without incurring extensive additional hardware development, a pair of hardware modules were developed, one performing computing and one the interfacing function. (See Fig. 5) Each is a printed circuit card carrying a number of sub-modules. The PCB tracks carry the interconnections always required in any application, while those connections needed for a specific task are added by component selection and appropriate wire links. The design for a computing card incorporates provision for up to eight microprocessors. All the analogue input and output lines from each processor, together with a substantial proportion of the edge connection lines are left uncommitted ready for suitable wire link patching to suit a specific task.

It is less obvious how the interfacing can be standardised. However, since the Intel 2920 has its own analogue Input/Output and the bulk of control signals will be analogue, the interfacing requirement reduces to a simple one of buffering, antialiasing filter, offset and gain adjustment. This list of requirements can be accomplished using a single op-amp circuit which can also act as the summing junction needed to perform the resolution enhancement referred to previously. Even the usually complex antialiasing filter can be accomplished with a simple first order filter because the very high sample rate relative to signal bandwidth will tolerate the shallow cut off slope. Thus a PC design comprising eight of these standard modules on a card has been produced and can be modified by suitable component selection to suit most tasks. The exceptions to this can be considered as special cases, and for this purpose an uncommitted area has been allowed on the PC card to cater for any special purpose circuitry.

4. DOCUMENTATION AND TESTING

The foregoing sections have shown that the software generation task can be reduced by functionally partitioning the hardware and software, and how this might be achieved in practice. However, this software still has to be free of errors and there is still a need for rigid specification, documentation, test and acceptance procedures. It is interesting to note the parallels between the computing philosophy outlined above and a general purpose analogue computer. This suggests a possible documentation and test philosophy based on conventional analogue practice which should ensure maximum visibility to all concerned (see Fig. 6).

Following design, analysis and simulation, control laws are normally specified by functional block diagrams. This form is readily converted into a full computer specification by identifying the function of each processing module and the details of the interconnections both between modules and external to the computer (including analogue signal levels, scale factors etc). Each module functional specification can then be converted into a program and thence into discrete hardware. The documentation of this module would comprise the program listing (fully commented), a definition of the scaling and truncation of all intermediate variables, and a definition of all possible context dependencies.

A test procedure is required for each module, independently derived from the module functional specification. This will be a hardware functional check to be performed on the processing module after programming by stimulating the inputs and monitoring the outputs. The tests will exercise all inputs, internal variables

and outputs over their full range of amplitude and frequency and will check for correct operation of all conditional instructions.

The test procedure serves both to verify the software and to check that the processor has been correctly programmed and is functioning correctly. Since the check out is fully comprehensive there is no need for separate software verification procedures based on emulations or other computer based procedures.

The complete computer would be functionally tested in a similar way, testing all input/output interfaces, all communications between modules, all modes of operation etc. but without having to repeat the exhaustive software checkout since no module can affect the correct functioning of any other module.

This reliance on functional specification and test promises to reduce the magnitude of the documentation task, and the increased visibility of the testing process should give improved confidence that the final product will operate consistently and correctly.

5. CURRENT APPLICATIONS

Of several applications being pursued, the most challenging involves the Royal Aircraft Establishment's Fly-by-Wire Hunter aircraft which is currently being operated jointly with British Aerospace, Brough, on a flight dynamics research programme. This aircraft is currently fitted with a quadruplex analogue active control system. It is hoped that by gradually introducing a number of processors into this system the multiple asynchronous microprocessor concept can be proved, while at the same time enhancing the capability (in terms of flexibility and complexity of control laws) of the existing system and it is intended that a number of the processing and interfacing cards be configured as a duplex, fail passive, computer which can be used for ad hoc extensions to the existing control law computations. Initially this will be of limited authority but, as confidence is gained in the system, more comprehensive control functions can be added with wider authority until all of the present analogue control law implementation has been replaced. This should provide a considerable increase in utility of this aircraft as an experimental vehicle, and should expose these asynchronous multiprocessor concepts to a realistic test of their practicality at an early stage. If this work is successful it is then anticipated that further extension of the digital computing sections could allow the failure management, built-in and pre-flight test functions (currently implemented with analogue techniques) to be updated until a multiprocessor configuration was achieved which would be fully representative of a production flight control computer configuration.

In addition to the above flight control applications there is a need to investigate the maintainability and survivability of such a system and its possible application to, and implications on, the rest of the aircraft systems. With this in mind a laboratory breadboard of a multiplex system is being developed to study the flight control system architecture per se and also to provide a means of emulating a flight control system to study the interaction with other systems. This work is therefore closely tied to other development work involving avionic and hydraulic systems rigs.

6. FUTURE DEVELOPMENTS

If, in pursuing these research tasks, the experimental system provides the flexibility and performance that is hoped for, then the next step would be to develop a fully engineered version. At that stage the choice of processing module would be reassessed in the light of experience, bearing in mind the less severe constraints on hardware development. In particular the choice of analogue communication between modules, while being expedient in the short term, is inappropriate for an engineered system since it reintroduces some of the problems of analogue systems: it is susceptible to noise pick up, gain variation and offset problems, and can have significant variation of characteristics with temperature. A possible alternative is to use a small dual port memory as an asynchronous buffer between each pair of processing modules. There would then be more freedom of choice of processing module and this could make other desirable features such as hardware multiply available. This type of development would result in a chip set rather than a single chip processing module. This could conveniently be integrated using a hybrid packaging technique to retain the circuit design advantages of a simple modular structure.

It is worth noting that this asynchronous multiprocessor concept with its very simple communication structure lends itself to investigation of other advanced flight control system concepts. In particular, the 2920 Signal Processor with its analogue interfaces should be eminently suitable for implementing hybrid dissimilar redundant control systems where a very simple analogue control loop is augmented by an advanced digital controller (such as that suggested by GILL F 1979). Also it has been suggested that asynchronous multiprocessors can be organised into a fault tolerant system by the addition of suitable control structures (ref. Segall et al 1979). While this observation was aimed at general purpose computing, a simple variant on the theme could allow an equivalent philosophy for dedicated control processing to be developed. The aim of these studies would be to reduce the level of redundancy required in order to achieve a high integrity control scheme. Both schemes operate by accepting degraded operation of non essential functions following a failure. If the level of redundancy required could be reduced, then it could allow the considerable benefits of active control techniques to be applied to a much wider range of aircraft.

7. CONCLUSIONS

Digital computation of control functions using a multi redundant system offers considerable benefits over a similar analogue system. However, it introduces some difficulties of its own, particularly 1) a lack of visibility of system operation which complicates testing, 2) time delays and synchronisation problems which complicate the control law design and the coding, 3) possible occurrence of obscure context dependent failures.

A multiprocessor flight control computer allows the software task to be partitioned into convenient modules thus easing the generation and testing of suitable code. It allows these modules to run in parallel thus reducing time delay problems. Asynchronous communication over dedicated links provides visibility of operation so aiding test and acceptance procedures. Finally, a restricted instruction set can substantially

reduce the number and type of possible context dependent problems.

Thus the task of developing and testing flight control software should be considerably eased. This is particularly important during the development phase of a new aircraft, or for an experimental control law/flight dynamics research aircraft. The disadvantages of this approach are that it does not provide a minimum hardware solution and it does not lend itself to high order matrix computation. These factors are probably not significant given the rapidly reducing costs of hardware and the control techniques which are likely to be used in aircraft in the foreseeable future. The asynchronous multiprocessor approach may even introduce hardware benefits by developing a number of modules which can be readily configured for a wide range of high reliability control applications.

If in total these factors reduce the software task to a level which can be supported "in house" then major improvements should be possible in the rate at which results can be achieved from and improvements incorporated into a flight development programme. In practice these potential advantages can only be assessed on the basis of practical experience and it is hoped that the research programme outlined above in both ground rig and airborne applications, will demonstrate these.

ACKNOWLEDGEMENTS

The authors are indebted to British Aerospace for permission to present this paper, and to the Royal Aircraft Establishment for their support and encouragement during the course of this project. The views presented, however, are entirely their own.

REFERENCES

- | | |
|---|--|
| Corney, J. M. | Aircraft active control systems: the inner loop. RAeS Spring Convention "Aerospce Electronics in the Next Two Decades" 1979. |
| Gill, F. | Ideas for future efficient flight control systems. RAE Tech. Memo. FS 256, 1979. |
| Jackson, K. and Simpson, H. T. | MASCOT - A modular approach to system construction operation and test. AGARD CP 149 1974. |
| Segall, Z., Yoeli, M., Strosbourger, E. | Parallel fault tolerant computation structure. Computers and Digital Techniques Vol. 2 No. 2. |
| Sharaz, A. | Implementation of Flight Control System constituents using an Analogue Processor. BAe Tech. Note YED 6981 1980. |
| Wright, S. M. and Fletcher, M. | A technique for improving the effective resolution of an A to D converter. BAe Tech. Note YEL 6351 April 1980. |
| Wright, S. M. | Implementation of log hyperbolic, trigonometric and exponential functions on an Intel 2920 signal processor. BAe Tech. Note YED 6986 1980. |

- Real Time Digital Processing of Analog Signals
- Nominal Signal Bandwidths from DC to 10KHz
- Digital Processing Accuracy and Stability
- Special Purpose Instruction Set for Signal Processing
- Twentyfive Bit Wide Data Word
- 400 ns Instruction Execution Time
- Multiple Analog Inputs (4) and Outputs (8)
- On-Chip Sample and Hold Circuits and D/A Converter
- On-Chip EPROM: User Programmable and UV Erasable
- On-Chip Scratch Pad Memory (40 Locations)
- Analog and/or TTL Output Waveforms, User Selectable
- 192 Program Locations

Fig.1a Summary of INTEL 2920 Processor Features

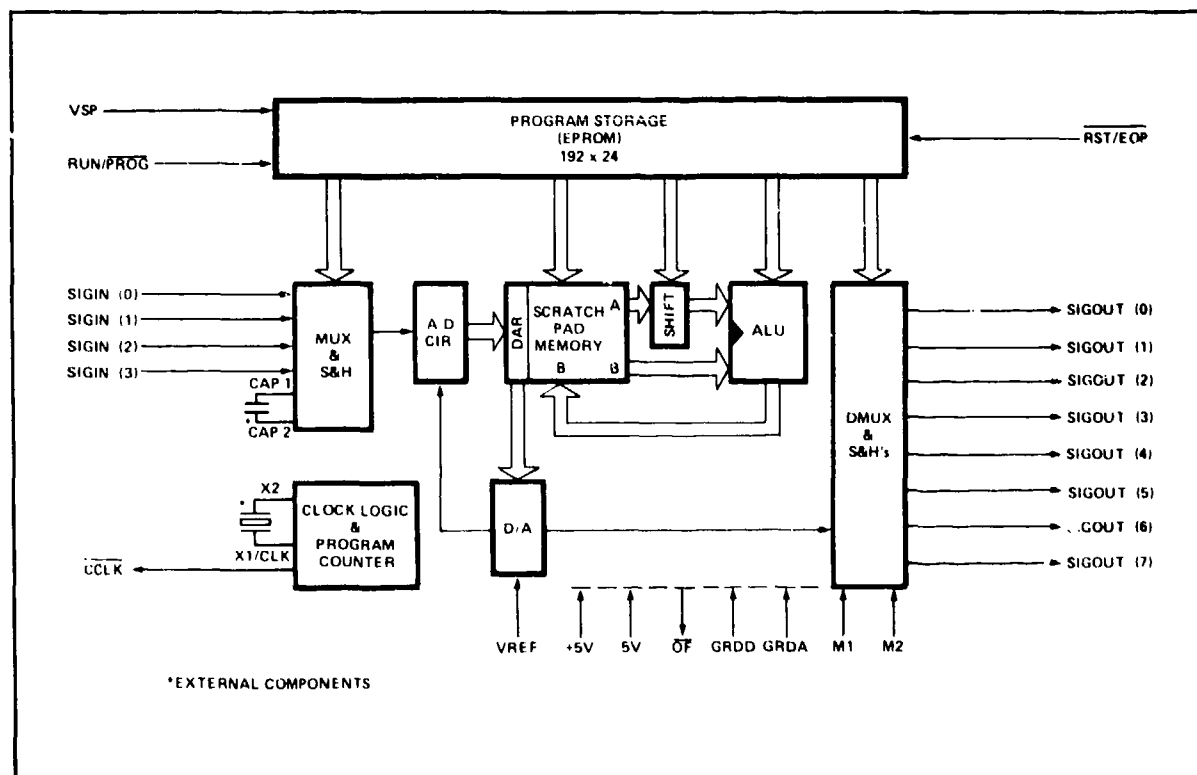


DIAGRAM COURTESY OF INTEL

Fig.1b Functional Block Diagram of INTEL 2920 Processor Architecture

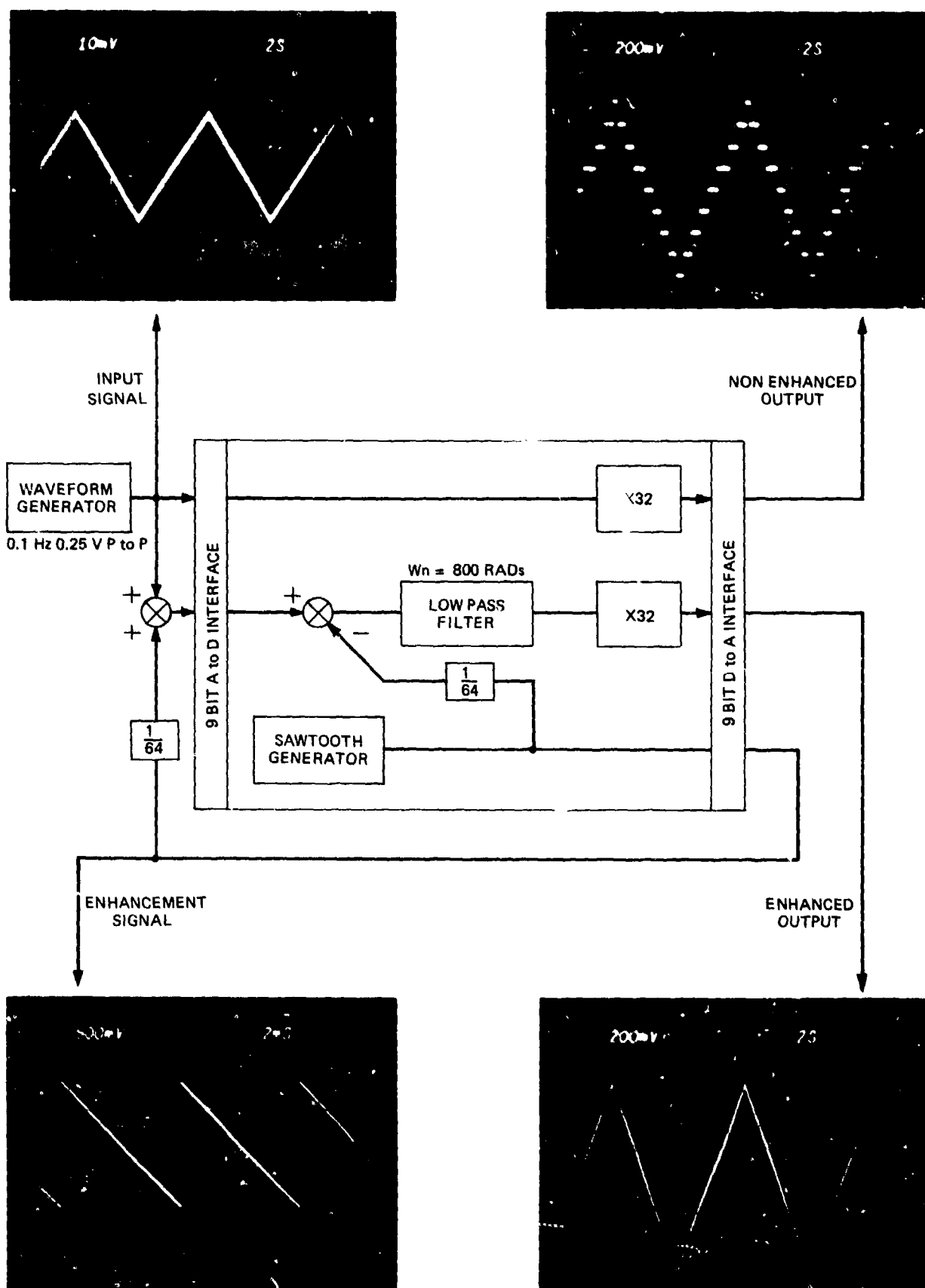
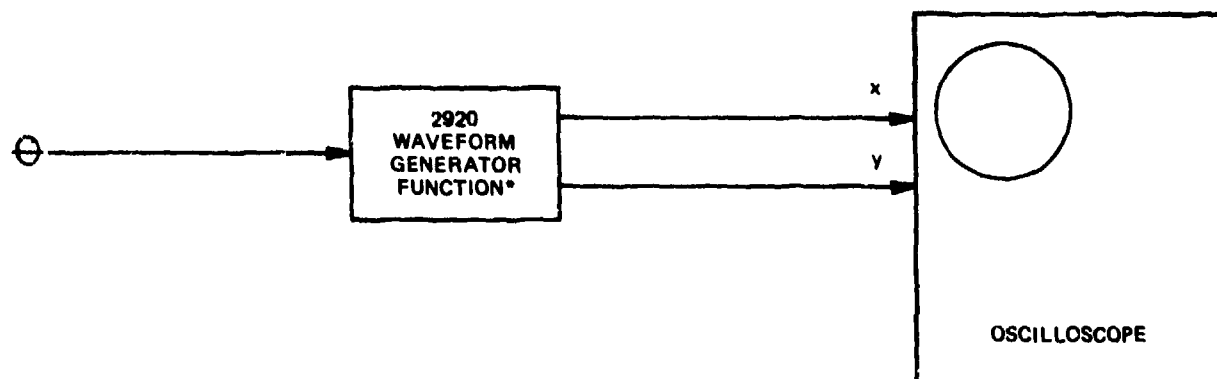


Fig.2 Resolution Enhancement Test Results



$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} A_0 B(t-t_0) & 0 & 0 \\ 0 & A_0 B(t-t_0) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(Wx t) \\ \sin(Wx t) \\ Cx t \end{bmatrix}$$

WHERE x, y ARE ORTHOGONAL INPUTS TO THE OSCILLOSCOPE
 A, B, C ARE REAL CONSTANTS

t = TIME

t_0 = INITIAL TIME

W = ANGULAR VELOCITY OF HELIX VECTOR ROTATION IN
 THE ORIGINAL x, y PLANE ($\theta = 0$)

θ = INPUT ANGLE USED TO ROTATE THE HELIX IN THE
 Z PLANE NORMAL TO THE ORIGINAL x, y PLANE

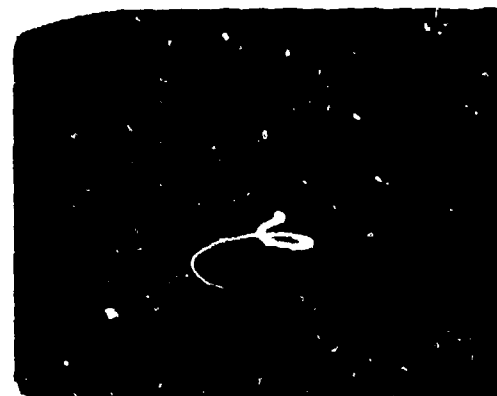


$\theta = 90^\circ$

TYPICAL DISPLAY
 REPETITION RATE 25 Hz

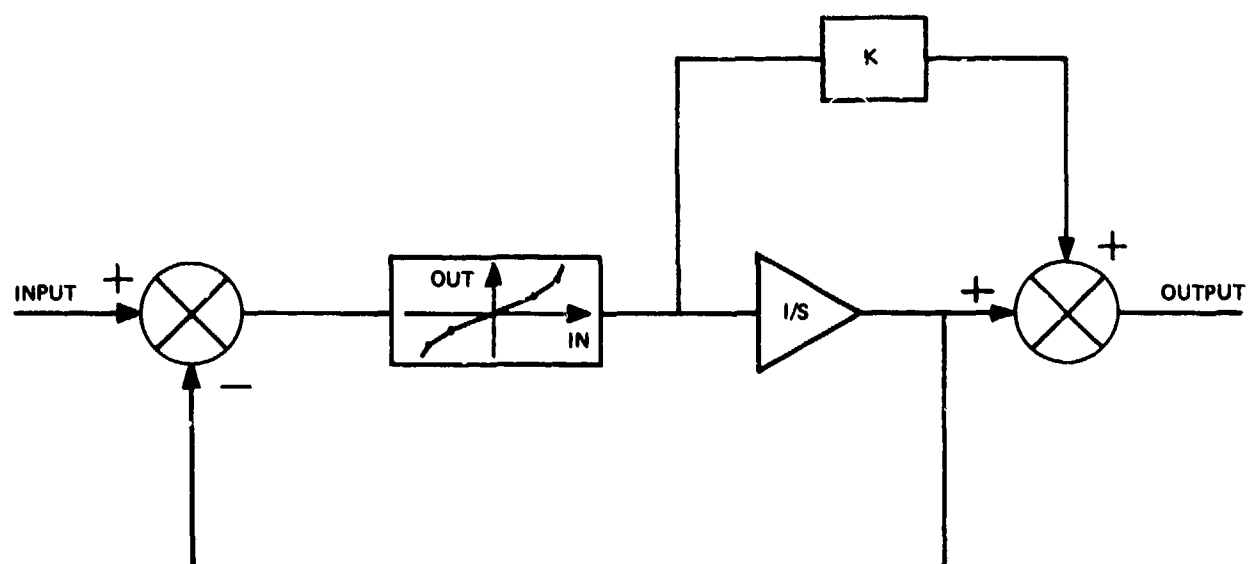


$\theta = 0^\circ$

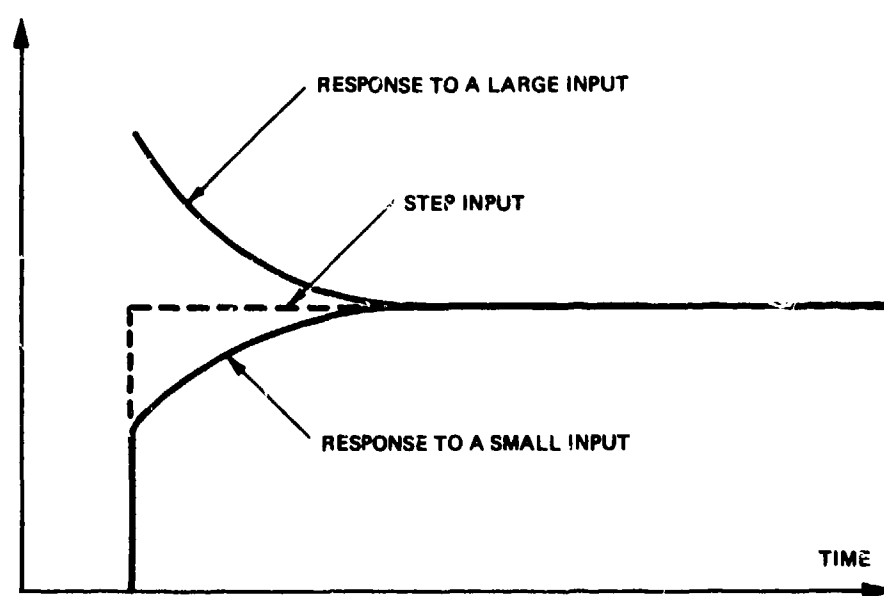


$\theta = 45^\circ$

Fig.3 Example of Trig Function Implemented on INTEL 2920



BLOCK DIAGRAM OF THE FILTER



COMPARISON OF NORMALISED RESPONSES TO A STEP UNIT

Fig.4 Non Linear Filter – In Example of the Type of Control Law Function Currently Implemented

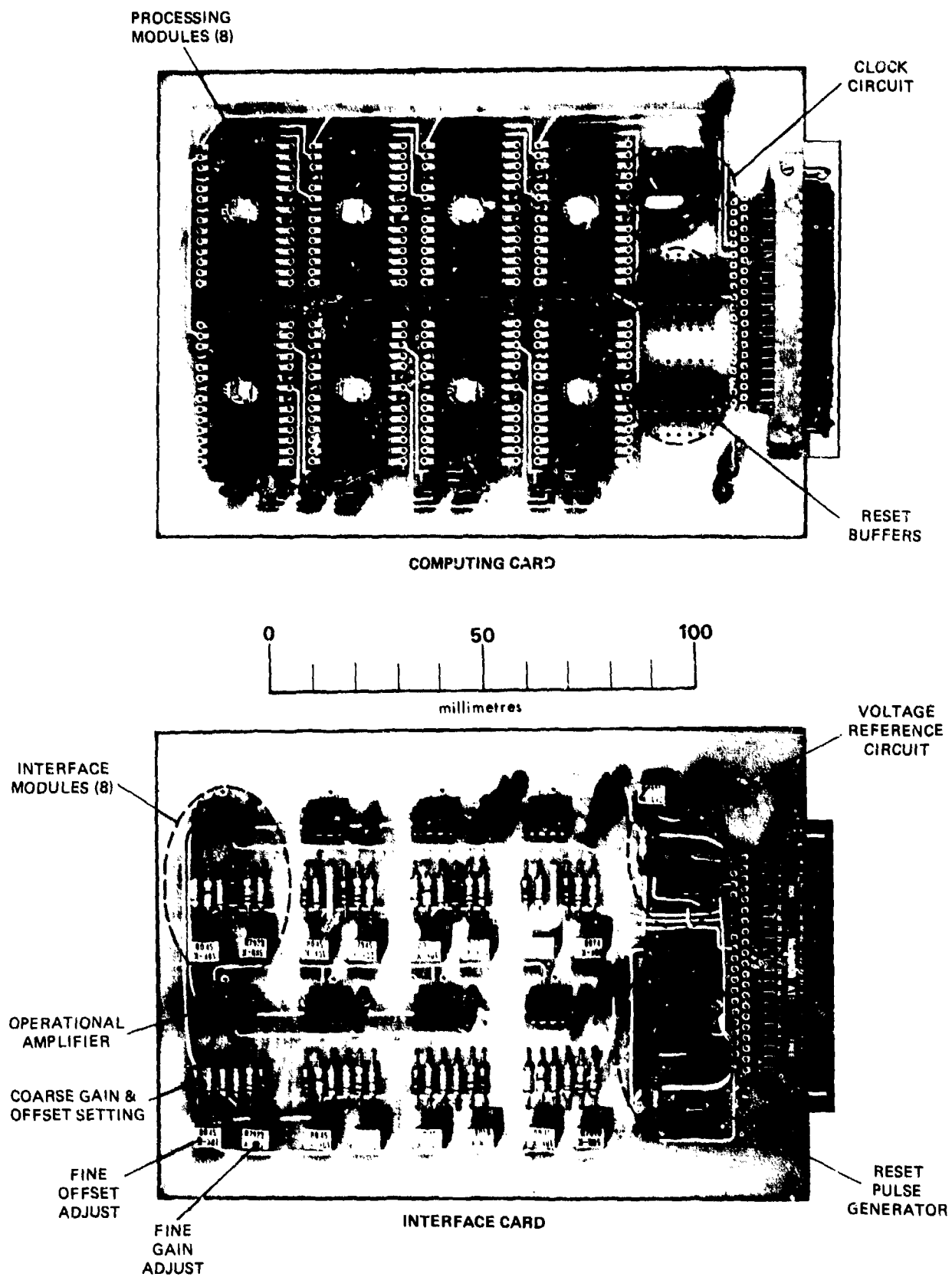


Fig.5 Example Computing and Interface Cards

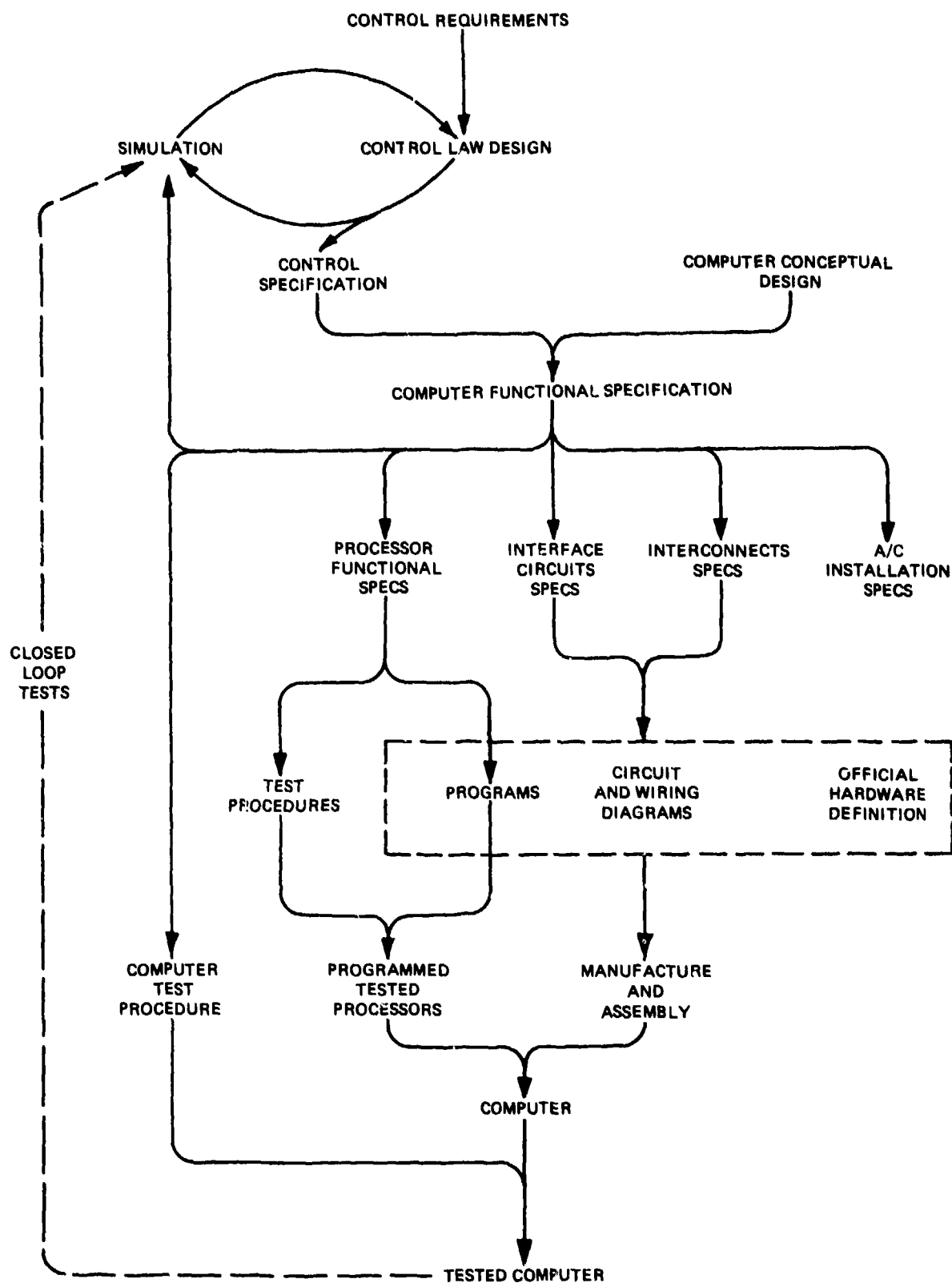


Fig.6 Design, Test and Documentation Process

FUNCTIONAL VERSUS COMMUNICATION STRUCTURES IN MODERN AVIONIC SYSTEMS

by

K. Brammer and A. Weimann
 ESG Elektronik-System-Gesellschaft mbH
 Postfach 800569
 D-8000 Muenchen 80
 W. Germany

SUMMARY

In the early design stages, an avionic system is functionally structured into subsystems, which in turn are broken down into functional units (equipments). With conventional technologies and with signal wiring connections of the single source, single drain type, the functional structure, which is of the hierarchical type, could more or less be carried over to the implementation stage. Especially the line replaceable units comprising an equipment were typically wired to the master unit of the equipment which in turn mainly communicated with the master unit (e.g. computer) of the subsystem.

In recent years this situation has been changing rapidly. Current technological trends that have major implications on avionic system structures are:

- For intrasystem signal transmission, networks of wires connecting a single transmitter with a single receiver are being replaced by bus systems with time division broadcast characteristics.
- Progress in data processing technology renders it feasible to assign digitally performed functions to much lower system levels than before.
- In aircraft design, control configured vehicle (CCV) technology implies the substitution of mechanical means for flight critical functions, such as basic stabilization and primary flight control by electronic data processing and transmitting means. This has raised unprecedented requirements on reliability and survivability of avionic elements and intrasystem communication.
- In the field of navigation sensors, mechanically stabilized units like inertial platforms, Doppler radar antennas, flux valves etc. are replaced by strap down sensors, where the decoupling of sensed information from the aircraft's rotations is now performed by electronic data processing.
- Scanning of directional sensors, e.g. fire control radar, ESM or ECM antennas, is increasingly performed by electronic means.

In the paper, the implications of the accompanying increase in functional and communication interfaces on avionic system structures are analyzed. Especially the passage from functional design to implemented communication structure of the airborne electronic system is scrutinized. The distributed organisation of an avionic system, the realization of which is greatly simplified by bus type intrasystem signal transmission, is compared to the conventional hierarchical system organisation. Advantages and drawbacks of both organisations are reviewed especially with respect to interface efficiency, cabling requirements and the typical topology of avionic systems.

The topic is illustrated by the structures of a conventional and a modern avionic system.

1. INTRODUCTION

The paper addresses a problem which has arisen in avionic system design due to technological changes in intrasystem communication. In the past, there existed a great degree of correspondence - at least in principle - between the process of functional structuring of an avionic system in the design stage on the one hand, and the communication structure within the system on the other hand. Both structures were essentially of the hierarchical type.

In the meantime the advent of new concepts and technologies has brought about a certain discrepancy between the functional design of the system and the implementation of intrasystem communication. Whereas the former continues to be hierarchical, the latter treats the terminals as peers.

It seems that this trend has been produced mainly by three developments: the simplification of cabling, e.g. by the use of bus systems, the distribution of processing to equipments and line replaceable units, and the transfer of network and switching concepts from telecommunications to computer networks and, subsequently, to avionics systems.

In this paper an attempt is made to draw a partial resumé of the former clean situation as a reference and to discuss the new mixed situation with respect to this background.

2. HIERARCHICAL ASPECTS IN AVIONIC SYSTEMS

2.1 Avionic System Design Principle

The basic method and the main steps of avionic system design have become fairly well settled and generally accepted. Here we sum up the major features as a starting point for the subsequent analysis.

The task of system design is always subject to the relevant general constraints such as national or international standards, practices, logistics procedures and so on. These are not always explicitly listed by the customer, rather their knowledge is often implicitly expected to be part of the professional experience of the designer.

The specification of the avionic system requirements is the basic document containing the technical points of reference for the system to be designed. It defines the task, the functions, the performance and the modes of the system, together with its technical boundary conditions (e.g. given constraints regarding weight), the physical operating environment, the external interfaces (e.g. communication, power supply, man machine interface) and the availability parameters.

In response to this input, the designer conceives and nominates the system parts which in combination are potentially able to fulfill the requirements.

The interrelation of these parts is then manifested by the design of the system architecture and organisation, i.e. by creating the structure and assigning functional responsibilities and management authorities to hardware parts, software parts and the operator.

This step must be accompanied by the definition of all arising interfaces between the system parts. Now the fulfilment of the requirements can be checked. If the result of this cycle is positive, one is able to specify the system parts.

Usually the decomposition of a system requirement or specification into a set of partial specifications is not done in a single cycle, but in repeated cycles at successively lower system levels.

Although in reality it is not always possible to follow this design procedure in complete purity, this so-called top down design philosophy has become widely accepted as a basic guideline.

2.2 Functional Architecture

Figure 1 illustrates the top down design process and the resulting functional architecture of the avionic system (LAUBER, 1980).

At the top level we have the functional description of the overall system. At level 2 the decomposition into functional areas or subsystems has been performed. The intermediate level between levels 1 and 2 describes the interrelations between the system and its subsystems and between the subsystems among each other.

The next cycle leads from the subsystem level to the level of functional modules, implemented either by software or by hardware, i.e. equipments.

From a systems engineering point of view it is necessary to proceed until the level of construction modules, at least in case of hardware, because the installation and power supply of all black boxes or line replaceable units must be defined.

The breakdown of black boxes internally, e.g. into circuit boards, is usually left to the equipment manufacturer and is of no concern in the following discussion.

It is evident from Figure 1, that the top down design method automatically produces a hierarchical set of specifications for the parts of the avionic system.

2.3 Interface Efficiency

It is remarkable, that one finds much agreement on the top down procedure, but scarcely any philosophical or useful theoretical justification for it. The feeling exists that it is an economical and efficient way to proceed.

In Fig. 2 this point is confirmed with respect to the maximum number of potential interfaces among the members of hierarchies as compared to peer groups.

As a reference we use the total number of possible mutual interfaces in a peer group. This number R is obviously equal to $N(N-1)/2$ where N is the number of members. The number of interfaces in a hierarchy is called R_H . Dividing R_H by the reference number R , we obtain a measure of interface efficiency (BRAMMER, 1981). This measure is plotted in Fig. 2 as a function of the number of members, N , in a double logarithmic scale. Two parameters are used to describe the hierarchy: the number of levels, and the number of associates to each master. For simplicity the latter parameter is kept equal for all masters, regardless of the levels.

If the hierarchy has only 2 levels, the number of all possible mutual interfaces is equal to the case of the peer group: the interface efficiency quotient remains at one.

In all other cases the hierarchy features less interfaces than the peer group. The interface efficiency improves uniformly and markedly along with the growing number of levels, with the shrinking number of associates and with the total number of members, as shown by the set of decreasing lines.

For example, consider a group with the order of 32 members. In the peer group or in a two-level hierarchy they have about 500 possible interfaces. The same number of members, organised in three hierarchical levels with 5 associates to each master, have only 20% or 100 possible interfaces. If they are organised in 5 levels with 2 associates, the number of interfaces reduces still further to 10%.

The efficiency effect is clear and uniform. It is the more marked, the larger the group of members is. For instance, with 1000 members, the number of interfaces in hierarchies with up to ten associates is 1% and less, compared to the unstructured case. Although the hierarchy has weaknesses in other respects, its interface efficiency can be judged as an advantage.

Clearly the number of interfaces is a measure for the labour involved in complete system specification down to component level, to contract negotiations, acceptance test and system integration activities.

2.4 Classical Communication Structure

In the classical avionic system, the implemented communication structure basically followed the hierarchical system organisation, see Fig. 3. This was essentially due to:

- the presence of a single central computer as the only resource for digital general purpose data processing,
- the prevalence of single source-single drain data and signal transmission lines, and
- the co-use of the central computer as a central message switching node in order to allow multi-user interconnections despite the absence of bus technology (CARRUTHERS, 1979).

For example, in classical avionic systems, the subsystem functions are centralised in the form of subprograms within the main computer. These subprograms communicate via dedicated links directly with the associated equipments. In Fig. 3 the equipments in the upper line belong to the navigation subsystem, the first four equipments in the lower line belong to the displays and controls subsystem, etc.

The wiring shown goes between the central computer and the master unit of each equipment. Their associated line replaceable units (black boxes) are in turn wired to the equipment master unit.

This way a hierarchical communication network, formed by point-to-point links, is realised, reflecting very well the functional specification tree.

Of course, also here, reality is not as pure as the idea. For reasons of reliability, damage resistance and speed the considered system has numerous additional cross connections which were skipped here.

3. CURRENT TRENDS INFLUENCING AVIONIC SYSTEM ARCHITECTURE

For several years the architecture of avionic systems has been changing. Fig. 4 illustrates some of the major contributing trends and their interrelationships.

3.1 Technology

The left hand side of Fig. 4 shows examples for relevant technological advances. In the field of sensors they are phased arrays and strap down components, in the area of intra-system transmission we had the advent of high reliability electronic links, and in the processing field, high speed switching elements and large scale integration are being introduced.

3.2 Concepts and Equipments

The center part of Fig. 4 presents a number of current concepts and equipments influencing systems architecture. To name some of them, we have for instance

- Abstract implementation of coordinate frames
- Fly-by-wire
- Active stabilization and electronic control
- Multiplexed transmission, and of course
- Microprocessors and -computers.

3.3 Impact on Systems

In the context of this paper, the given factors have three main impacts on avionic systems, shown on the right hand side of Fig. 4. They are

- A substantial increase in signal and data processing
- Time division multi-source multi-sink transmission
- Locally distributed computing.

All three points have led to important structural changes: On the one hand, distributed computing allows location of processing functions at their proper level and frees the designer from concentrating them artificially in one single computer, see e.g. (SYRBE, 1978), (CIMS, 1979) or (BRAMMER, 1980). For example, the navigation subprogram can be removed from the central computer and allocated to a navigation subsystems computer. This type of distributed computing tends to spread out the functional hierarchy more visibly throughout the system topology.

On the other hand, distributed data processing in the strict sense implies not only physical dislocation of processing functions and associated hardware, but also distribution of the data base and of the control function (ENSLOW, 1978), (SCHERR, 1978). This philosophy tends to diminish the hierarchical features of system organisation. Furthermore, the transfer of network and switching concepts from telecommunications to computer networks (WECKER, 1979) and from there to avionic systems gives rise to peer-like communication procedures. Finally, multiple access, broadcast type transmission systems render economic implementation of direct all-to-all communication feasible.

4. COMMUNICATION IN AVIONIC SYSTEMS

4.1 Available Communication Structures

The communication structures available for avionic systems today are summarised in Fig. 5. Each line in the structures represents a connecting cable. A simplex connection contains one basic channel of the type shown top left, consisting of a transmitter, driver, line and receiver. A full duplex connection contains two such channels in opposite directions. Half duplex connections use the same line for both directions.

Every available structure shown allows the direct or indirect communication among all participating units, indicated as solid dots arranged in a circle.

Using conventional point-to-point links, usually bit-serial and word-serial, one obtains first the classical structures:

- Network of direct all-to-all connections
- the star
- the layered star.

The bottom part of Fig. 5 shows the newer structures using links with broadcast capability:

- the matrix formed by a set of single-source, multiple-sink channels, e.g. of ARINC 429 standard ("DITS")
- the multiple access bus, e.g. of MIL 1553 standard ("MUX"), carrying multiple-source, multiple-sink traffic in both directions on a time division basis.

4.2 Cable Lengths

Suppose that all the structures shown in Fig. 5 are implemented with links of the same technological state of the art, especially with the same serially transmitted data bit rate. Remember further that all structures allow messages to be transmitted from each unit to any other unit. Then, the main advantage of the bus structure above all the other structures is the minimum cable length. This is evaluated in Fig. 6 and compared to the cable length of the layered star, the star and the all-to-all network (BRAMMER, 1981).

For simplicity and generality, the topology of participating units has been assumed here as a uniform distribution at the points of a square raster with constant raster width in both orthogonal directions.

The graph shows the total cable length necessary to allow complete communication among all units. This length is normalised by the raster width and plotted against the total number of units on a double logarithmic scale.

For instance, for 20 units our model yields a total cable length of 450 times the raster width for the all-to-all network, as compared to 19 for the bus. The cabling efficiency of the bus gets even better for larger numbers of units.

Note however, that the star and especially the layered star are doing fairly well in this respect, too.

4.3 Topological Considerations

We have seen that from a functional point of view the layered star structure is the most natural. In Fig. 7 this is case A, shown top left in idealised form. In this example, there are two subsystems: One constituted by round units in the upper half and the other consisting of square units in the lower half. Each subsystem, in turn, has three equipments. Each equipment has a master unit and three associated units.

Nowadays, we can assume computing functions down to the equipment level. Then this topology represents a federated computer architecture, where distributed computing is allocated according to disjunct topological areas.

However, in a real avionic system the functional and topological ordering of the line replaceable units does not coincide as in case A, but is mixed up as in case B. The cabling pattern then no longer follows the layered star, but is better characterised as a superposition of several stars. So the advantage in cable length of the layered star cannot be realised.

The same mixed configuration of LRU's as in case B is shown in case C and it is obvious that from the cabling point of view the bus structure is not affected by the mixed topology of functional units. But the question is, whether it is really desirable that a single-level bus connects all units down to LRU level.

4.4 Modern Communication Structure

From Fig. 8 which represents a typical interconnection structure of a modern fighter avionics system, one can conclude that not each and every black box is connected to a common bus. The avionics bus - duplex for redundancy - picks up the subsystems such as navigation, fire control, flight control and some equipments that have many communication interfaces such as air data, multifunction keyboards and displays.

Thus, the present state of the art in avionic systems still features hierarchical levels of communication: the central system control, the subsystem computers and some equipments communicate on an upper level bus, while in the lower levels either dedicated buses (triplex for flight control) or even still star type cables are used.

5. CONCLUDING REMARKS

5.1 Advantages of Multi-Level Communications

It has been noted that a common single-level bus running past all units of the system has the minimum possible cable length of all communication structures. Nevertheless, a multi-level structure persists due to the following advantages:

- Hierarchical structuring is efficient, not only in the design process, but also for contractual specifications, configuration control, acceptance testing, integration, maintenance and retrofit.
- This efficiency is mainly due to the reduction of the various sorts of interfaces between units, especially the communications interfaces.
- Generally the data rate decreases when we pass from lower to higher levels, therefore transmission capacity problems are alleviated by layering.
- Vice versa, reliability requirements often differ among subsystems, giving rise to dedicated components and links.
- Functional autonomy of equipments is maintained if they have dedicated lines to their LRU's. Otherwise, equipment development and acceptance testing would be greatly complicated.

These points call for at least two levels of communication: System bus, and links between the LRU's constituting an equipment. An intermediate third level may be adequate for some subsystems such as flight control.

5.2 Characteristics of Distributed Processing

Distributed processing has become cost-effective and is increasing in avionic systems. The advantages are

- The hierarchical decomposition of subsystem functions can be directly implemented, yielding a set of smaller programs instead of one large central program.
- Autonomy of subsystems is possible, with better reliability and survivability characteristics.
- In conjunction with the use of communication buses the central computer is eliminated as a central node or switching element.
- Locally dispersed computing resources with reconfiguration capability are reducing vulnerability.

However, due to communication delays, the interplay of distributed algorithms is less deterministic than in the centralized case in that each part must operate without a complete instantaneous knowledge of the state of all other parts.

Furthermore, even in a distributed system of avionics application programs it is necessary that their functional authority and the validity of data bases be system-wide managed.

5.3 Remaining Problems and Outlook

We have seen that at the present state of the art we live with an - at least partial - discrepancy between functional and data flow structures in avionic systems.

Bus systems make a logical connection of all-to-all type easily feasible, allowing multiple use of sensors for improved system performance and/or distribution of processing resources for better failure or damage resistance. But, even if we restrict this to equipment level and above, the system design has to cope with a substantial growth in communication interfaces. The overlay of functions versus communications must be subject to careful book-keeping, timing and control. This problem is aggravated by dynamic reconfiguration capability of the functional system architecture, especially when time-critical, high-priority functions require a high degree of confidence to be served at the right moment without delay.

Regarding avionic system operation we note the persistence of three types of central system elements

- System functions synthesizing top level applications on the basis of subsystem functions
- Control of distributed data processing
- Control of bus transmissions

These elements remain critical and need special redundancy protection and installation considerations.

Summing up briefly, it might be suggested that for avionic systems the conflicting goals of deterministic system behaviour requiring few functional and communication interfaces and tight control on the one hand, and of enhanced availability requiring distribution of resources, reallocation of functions and many communication interfaces on the other hand, require more research and practical experience in order to harmonise them and to establish new adequate and generally accepted avionic system implementation procedures.

6. REFERENCES

- BRAMMER, K., 1980, "Architecture of Flight Guidance and Control Systems," Working Paper prepared for AGARD GCP WG 05, Functional Integration of Positioning and Guidance and Control Systems, ESG, Munich, 24 March 1980.
- BRAMMER, K., 1981 "Functional Interrelations and Communications Interconnections in Avionic System Structures," Tech. Rep. ES-T/81, ESG, Munich, 30 April 1981.
- CARRUTHERS, J.F., 1979, "SHINPADS - A New Ship Integration Concept," Naval Engineers Journal, April 1979, pp. 155-163.
- CIMSA, 1979, "On-Board Computer Systems: Architecture, Technology, Support Software," Topic 3 of Vol. V (Data Processing) of Initial Technological Studies on European Air Traffic Management, European Community, Brussels, Dec. 1979.
- ENSLow, P.H., 1978, "What is a Distributed Data Processing System?", Computer, Jan, 1978, pp. 13-21.
- LAUBER, R. (Ed.), 1980 "Einführung in das Entwurfs-unterstützende Prozeß-Orientierte Spezifikationssystem EPOS 80," Inst. f. Regelungstechnik u. Prozeßautomatisierung, Univ. Stuttgart.
- SCHERR, A.L., 1978, "Distributed Data Processing," IBM Syst. J., Vol. 17, No. 4, pp. 324-342.
- SYRBE, M., 1978, "Basic Principles of Advanced Process Control System Structures and a Realisation with Optical-fibre-coupled Distributed Microcomputers," Proc. 7th. IFAC Congress (Helsinki, June 1978), pp. 393-401, Pergamon Press.
- WECKER, S., 1979, "Computer Network Architectures," Computer, Sept. 1979, pp. 58-72.

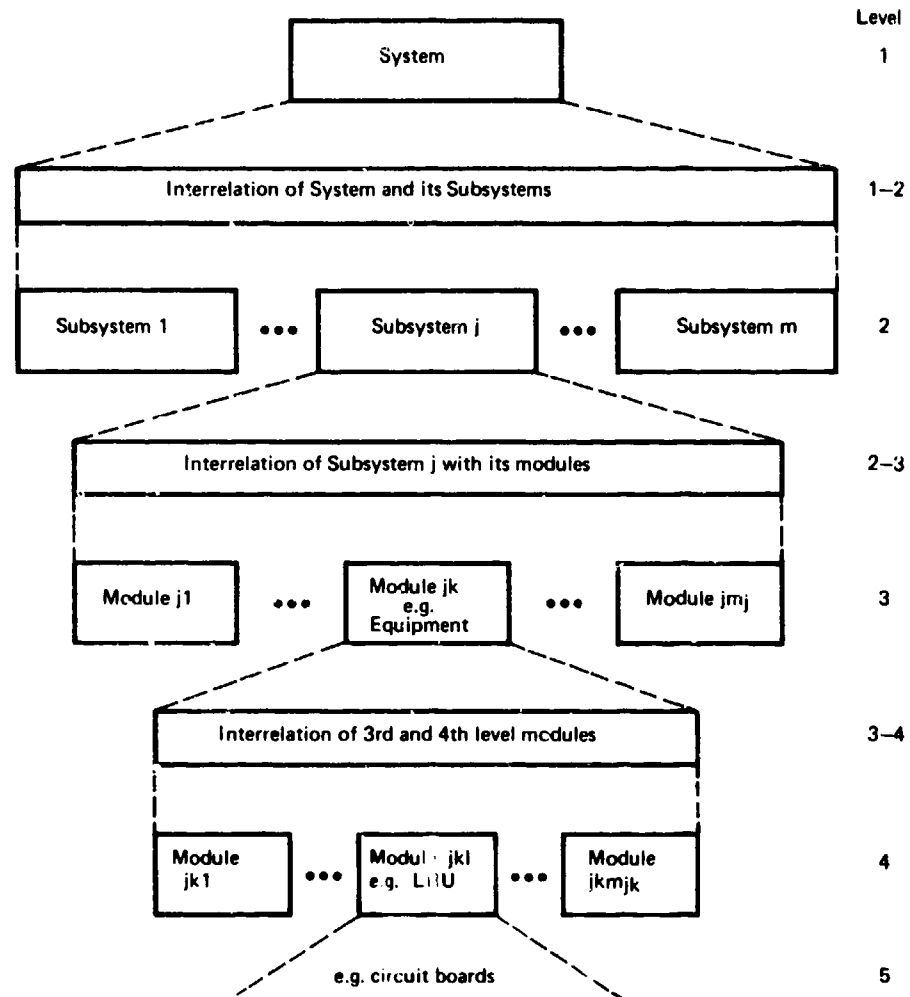


Figure 1: Hierarchical Description of Avionic System

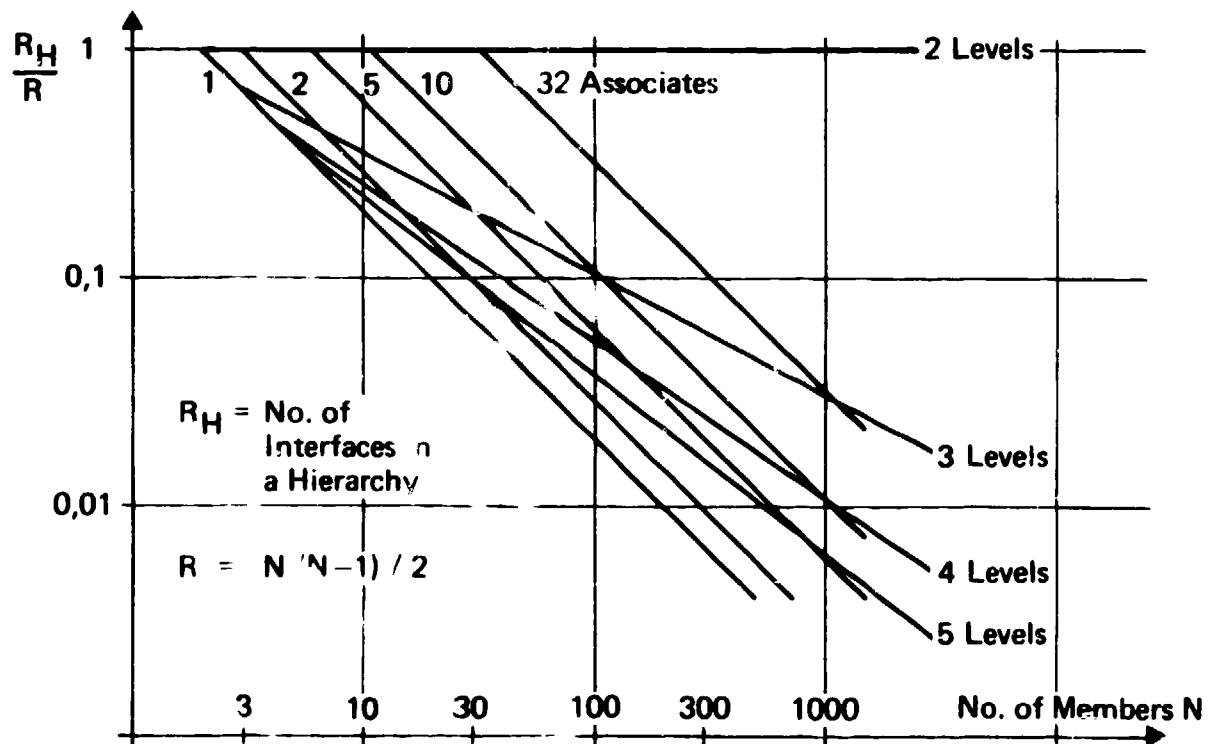
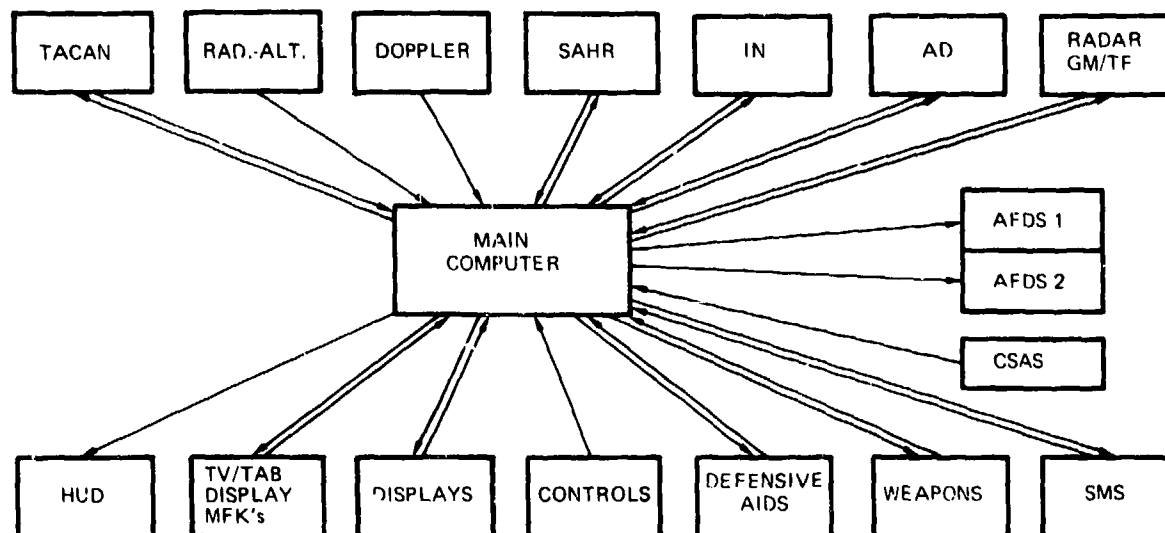


Figure 2: Interface Efficiency in a Hierarchy



Without wiring between Equipments and/or LRU's

Figure 3: Classical Avionics Interconnection Structure

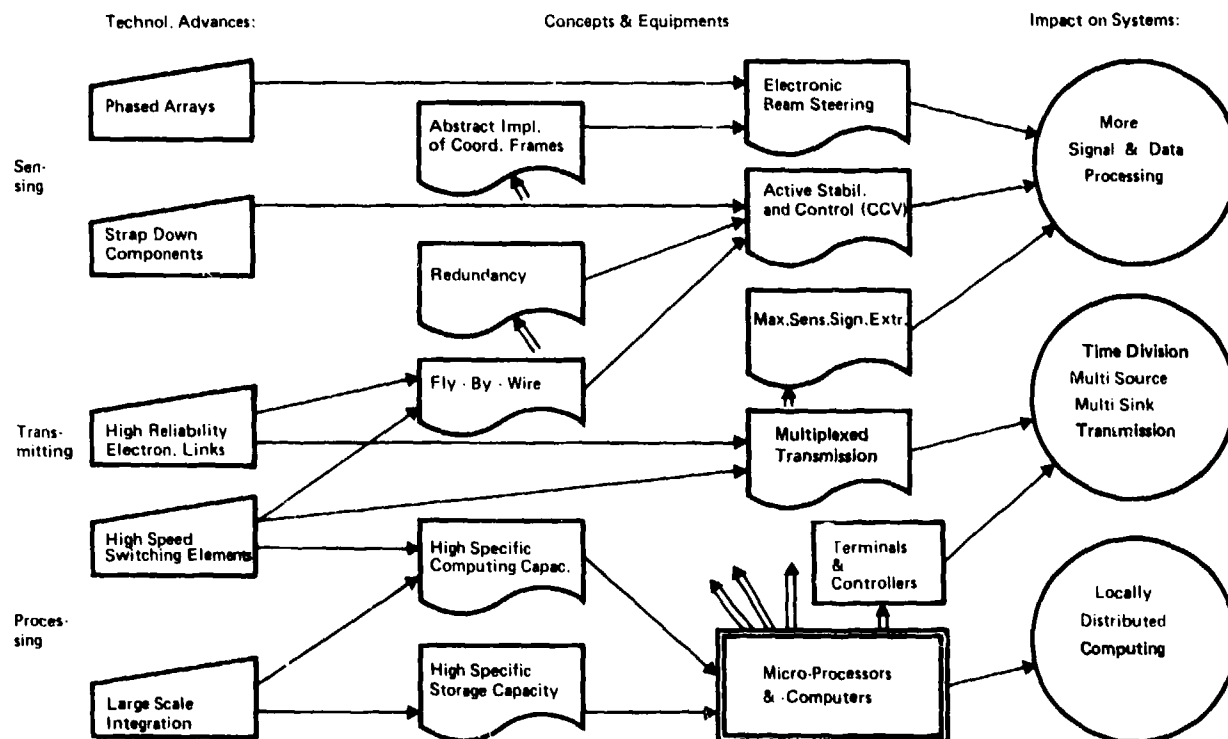
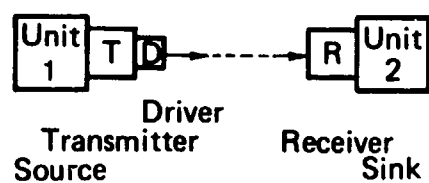
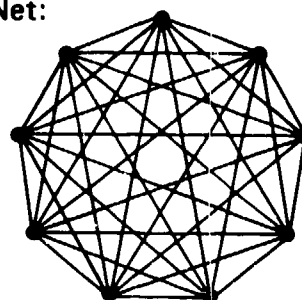


Figure 4: Current Trends in Avionic Systems

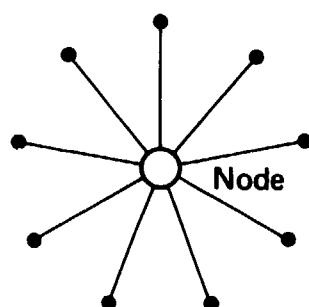
Basic Unidirectional Channel:



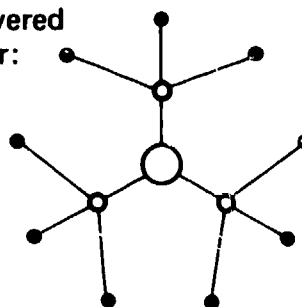
Net:



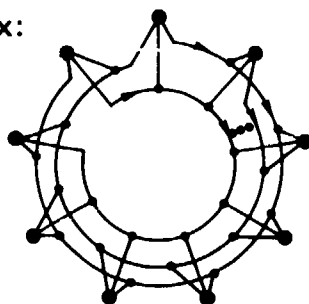
Star:



Layered Star:



Matrix:



Bus:

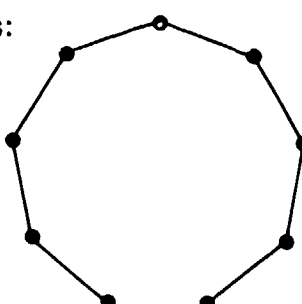


Figure 5: Communication Structures

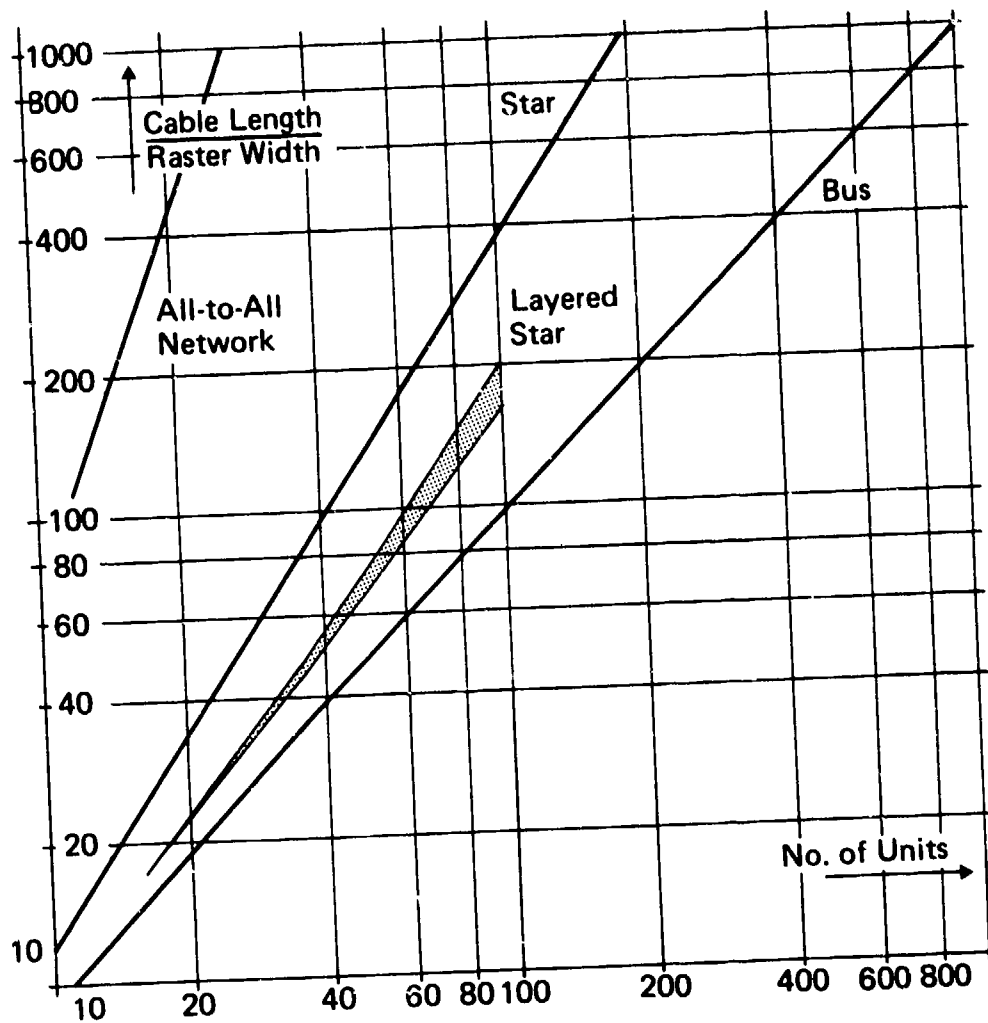


Figure 6: Total Cable Length in Square Rasters

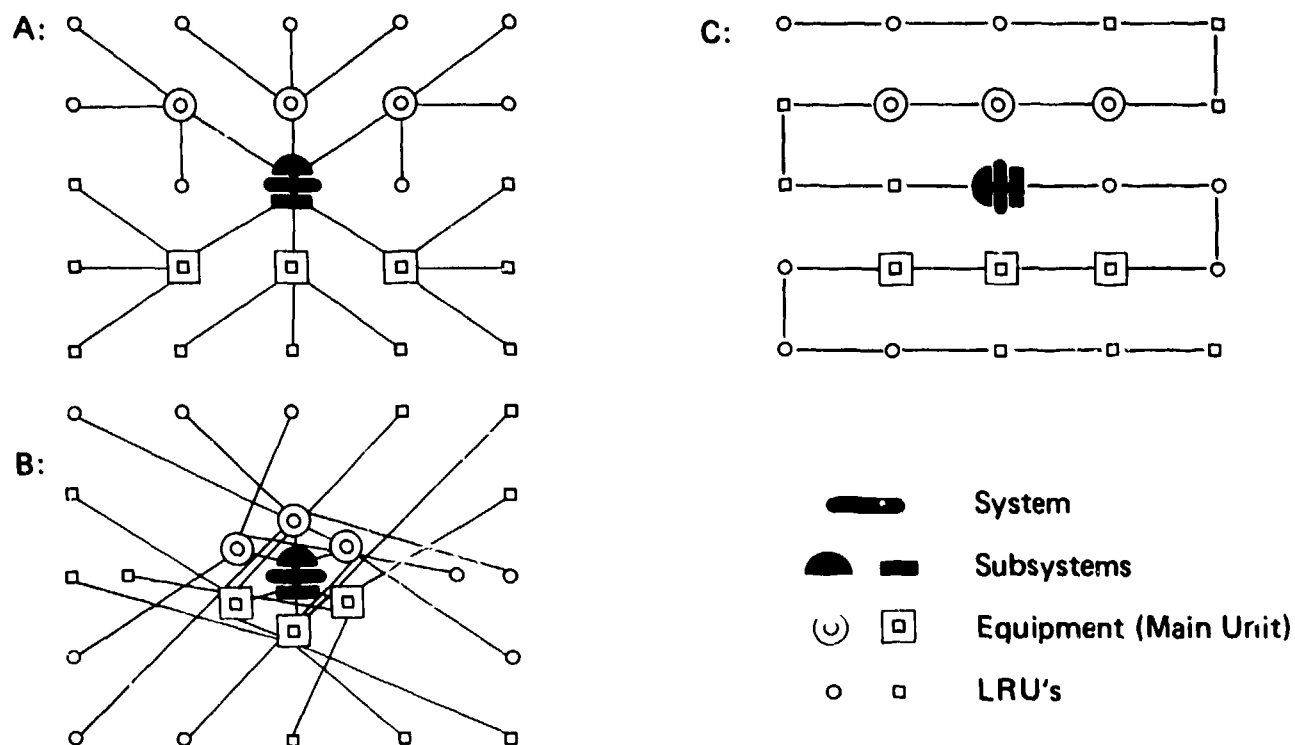
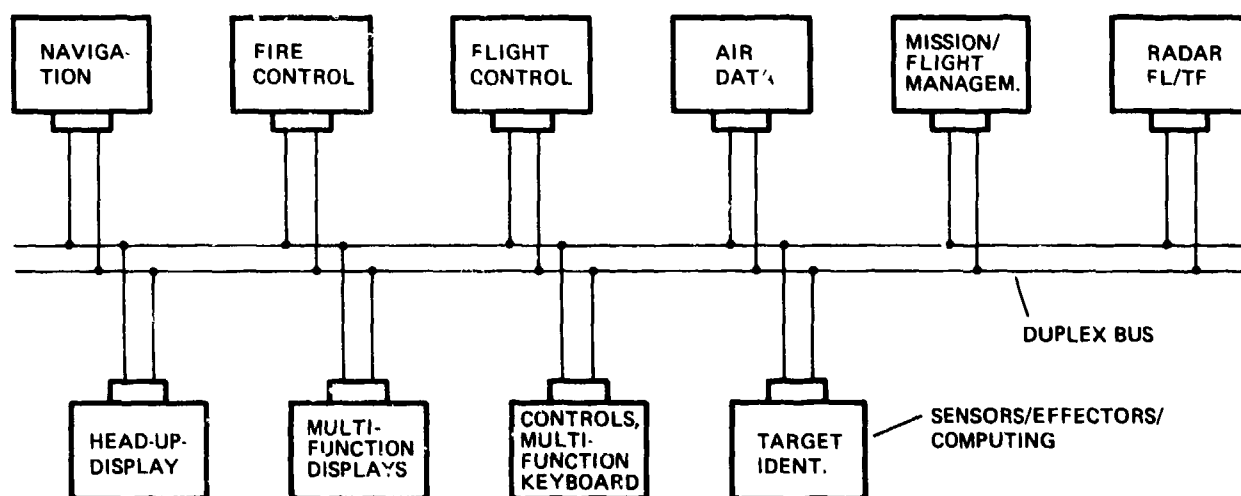


Figure 7: Topological Considerations



Without dedicated hardwiring (e.g. for video signals) and subbuses (e.g. flight control bus)

Figure 8: Modern Avionics Interconnection Structure

CONTINUOUS RECONFIGURATION IN A MULTI-MICROPROCESSOR FLIGHT CONTROL SYSTEM

LT. SCOTT L. MAHER AND CAPT. STANLEY J. LARIMER
Air Force Wright Aeronautical Laboratories
Flight Dynamics Laboratory
Wright-Patterson AFB, OH
U.S.A.

SUMMARY

Recent research at the US Air Force Wright Aeronautical Laboratories (Flight Dynamics Lab) has resulted in the development of a promising microprocessor based flight control system design. This system is characterized by a collection of cooperatively autonomous distributed microcomputers interconnected by an arbitrary number of common serial multiplex busses. Each processor in the system independently determines its assignments using a simple algorithm that dynamically redistributes system functions from processor to processor in a never-ending process of reconfiguration. This approach offers several potential benefits in terms of system reliability, and the architecture in general incorporates many state-of-the-art features which promise improved system throughput, expandability, and above all, ease of programming.

The Continuously Reconfiguring Multi-Microprocessor Flight Control System (CRM2FCS) represents a significant data point in multi-processor control system research. Promising ideas from a variety of references have been included and integrated in its design. Its laboratory implementation will provide a demonstration of the extent to which these ideas may improve throughput, reliability, and ease of programming in flight control applications.

1. INTRODUCTION

Before beginning a detailed discussion of the Continuously Reconfiguring Multi-Microprocessor Flight Control System (CRM2FCS) it is desirable to briefly discuss the design goals and philosophy which lead to this architecture. The original objective of this in-house effort was to develop an Air Force understanding of and capability in the area of multi-microprocessor flight control systems. It was determined that a high risk-high payoff approach could be taken in an effort to advance the state-of-the-art while achieving the primary objective. The approach taken was simply to make a trade off between low cost hardware and simplification of software as well as to distribute control to its extreme in an effort to obtain data as to the extent to which the potential advantages of such a system could be achieved. Other goals were to reduce overall hardware, software, and life cycle costs of flight control systems while maintaining high reliability and fault tolerance. Design considerations also included expandability for integrated control applications and reconfigurability to meet future self-healing requirements.

The concept of continuous reconfiguration is developed in some detail in this paper. An example is given and the advantages of such a scheme are discussed briefly. Autonomous control is introduced as an ideal method for controlling the continuously reconfiguring architecture. The requirements of a continuously reconfiguring autonomously controlled multi-processor architecture are listed and a novel bus contention scheme and the concept of virtual common memory are put forward as the means of meeting the requirements. Methods for simplifying software programming are also discussed as well as a description of a software simulation of the CRM2FCS. Finally the actual laboratory implementation of the architecture and the testing and data gathering facility to support the architecture are described.

2. THE CONCEPT OF CONTINUOUS RECONFIGURATION

Continuous reconfiguration is defined as a scheme whereby the tasks to be performed in a multi-processor system are dynamically redistributed among all functioning processors at or near the minor frame rate of the overall system. This approach allows continuous spare checkout, latent fault protection, and elimination of failure transients due to reconfiguration delay. By treating reconfiguration as the norm rather than the exception, failures can be handled routinely rather than as emergencies, resulting in predictable failure mode behavior. Using this approach, it is projected that the need for unscheduled system maintenance may be greatly reduced.

2.1 Example Of Continuous Reconfiguration

An example of what is meant by continuous reconfiguration is shown in Figure 1. A system of 9 processors is shown performing 6 different tasks, A thru F during three consecutive time frames. During the first time frame processor 1 is doing task B, processor 2 task D, processor 3 is a spare, and so on. In continuous reconfiguration the tasks are redistributed among the processors at the beginning of every time frame. For example, in the second time frame, there is an entirely different assignment of tasks to the processors. This reassignment is accomplished by having all of the processors that are currently healthy in the system compete for task assignments. If a processor fails during any time frame, it is no longer able to compete for task assignments. In Figure 1, if processor 4 failed during the second time frame, then during the next frame, it would not be able to compete for task assignment. The 6 tasks which need to be done are taken by healthy processors and the 2 remaining processors become spares. In other words, a

failed processor simply disappears from the system without any other processors being aware that it is gone.

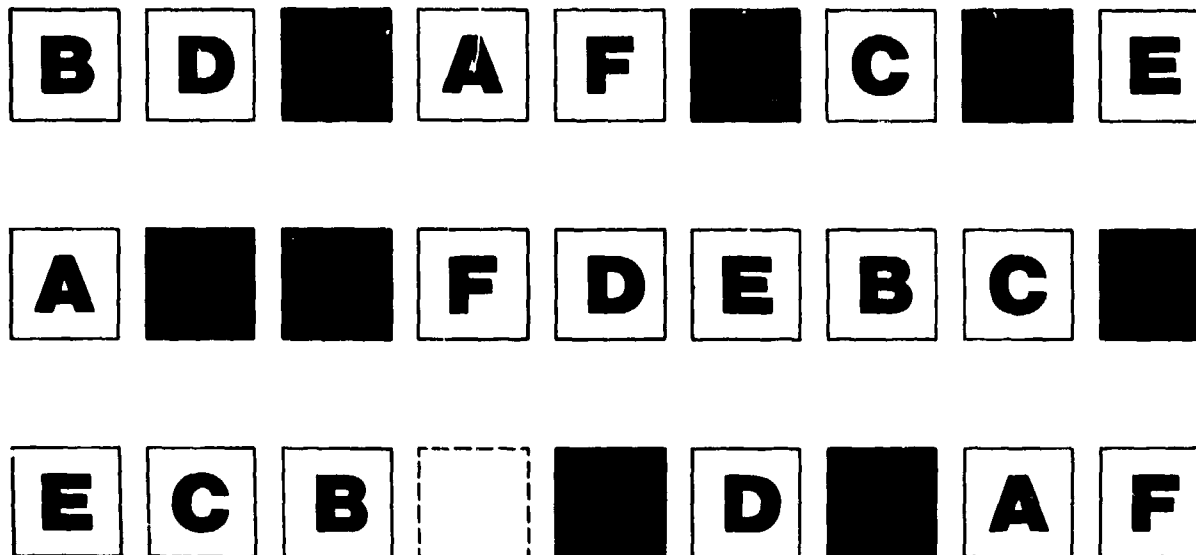


Fig. 1 Continuous Reconfiguration

2.2 Advantages To Continuous Reconfiguration

There are a number of advantages to the continuous reconfiguration approach. One of these is the ability to have continuous spare check-out. In traditional systems, where certain processors are permanently assigned to the spare status until they are needed, it is possible for one of these processors to fail while functioning as a spare. When a system processor fails and the failed spare is brought on line, catastrophic results may occur. The technique of continuously switching which processors are acting as spares allows every processor in the system to be constantly exercised. If a processor does fail, it is identified quickly and removed from the system, before it can cause any problems.

Latent fault protection is another advantage of the continuous reconfiguration approach. Latent faults are a class of faults that are characterized by the partial failure of a processor. The processor failure is not immediately detectable and may impede the systems ability to recover from any subsequent failures. Continuously exercising each processor, so that over a period of time every processor performs every task, forces a partially failed processor to reveal its failure and be removed from the system before it can interact with another partially failed processor in a manner that may preclude recovery.

A third benefit of continuous reconfiguration is zero reconfiguration delay. Most systems that are reconfigurable treat a failure as an emergency requiring special processing. This produces delays and possible failure transients in bringing the system back to its fully operational state. With continuous reconfiguration there is no emergency. The system reconfigures naturally every time frame so that, when a failure occurs, the system takes it in stride and with no failure transient.

2.3 Controlling A Continuously Reconfiguring System

A unique approach has been taken to controlling the continuously reconfiguring multi-microprocessor flight control system. One approach would be to have a central controller in charge of assigning tasks, handling reconfiguration and controlling bus access. A high throughput computer would be needed to meet the overhead requirements of the continuously reconfiguring architecture. A central controller also introduces the possibility of a single point failure in the system requiring redundancy incompatible with the architecture and reducing the reliability of the continuous reconfiguration concept.

An alternative approach to a central controller is autonomous control. This is a scheme whereby each processor independently determines its own next task based upon the current aircraft state. This can be better understood by using an analogy. Like the traditional centrally controlled computer architecture, a company has a president who has several vice-presidents working for him. The president has access to all information concerning the states of the company and an understanding of how the company should function. He uses this knowledge to allocate tasks to the vice-presidents and arbitrate any disagreements that may arise between them. Autonomous control is analogous to replacing each of the vice-presidents with a clone of the president. The vice-presidents are now capable of making the same decisions that the president would have made under the same circumstances, since they have access to the data that he had and would go through the same decision making process that he would. The need for the president has been eliminated and he has been replaced by autonomous vice-presidents. This approach is not practical in the human world because no two humans think alike. In the computer world, however, it is a realizable possibility.

2.4 Requirements Of A Continuously Reconfiguring System

In order to make continuous reconfiguration of autonomously controlled processors possible, several requirements must be satisfied. These requirements include an efficient bus contention scheme, availability of system state information to all processors, availability of all software to every processor, and a well-defined set of task assignment rules. The methods used to meet each of these requirements in the laboratory implementation are covered in some detail. Considerable attention has also been devoted to techniques for simplifying the actual software design for use in this system. Such a scheme is clearly required if the organization of a large number of processors, performing complex flight control algorithms, is to be implemented without total chaos. The two-dimensional task assignment chart is introduced to simplify this process.

The first requirement is for a set of well defined task assignment rules. Each of the processors must have an efficient means of determining the next task that it is required to do. There must not be an opportunity for any processor to conflict with other processors in the system and cause system failures. The task assignment rules are a function of the operating system software (Larimer, S.J., JUNE, 1981) and are discussed further in section 5.

A second requirement is that all processors must have all software. In order for a processor to be capable of doing any system task at any point in time, it must have the software available to do the task. This may seem unrealistic at first but a study of the trends in memory technology reveal that memory will continue to double in density every year to year and a half for at least five years and that the cost of memory will continue to go down. This trend makes supplying all software to every processor a reasonable trade to get the benefits offered by the CRM2FCS.

A third requirement of this system is that all processors must have all data. A processor must be capable of doing any task at any point in time and in order to perform most tasks must have access to data concerning the present state of the aircraft. This requirement could be met almost ideally by the common memory architecture illustrated in Figure 2b. The common memory is accessed equally by every processor in the system. This is excellent from a software standpoint, since the programmer can treat the common memory as though it were a part of the processor's local memory. Simply reading variables from a set location and writing results into other locations.

Although ideal from a software standpoint it is very poor from a hardware standpoint. The number of processors that can access the common memory is limited to the number of ports which can realistically be interfaced to it. This approach also introduces complex timing problems when more than one processor attempts to access the common memory at the same time.

A more suitable architecture from the hardware standpoint is the common bus structure also shown in Figure 2a. The processors in this architecture are interconnected by a common serial bus. The number of processors that can be attached to this bus is virtually unlimited and the interface hardware is relatively simple. This is a poor architecture from a software standpoint, however, since data must be formatted before transmitting it and must be processed as it is received. This architecture is also subject to bus contention problems when more than one processor attempts to transmit data on the bus simultaneously. The fourth requirement is, therefore, that an efficient bus contention scheme is needed.

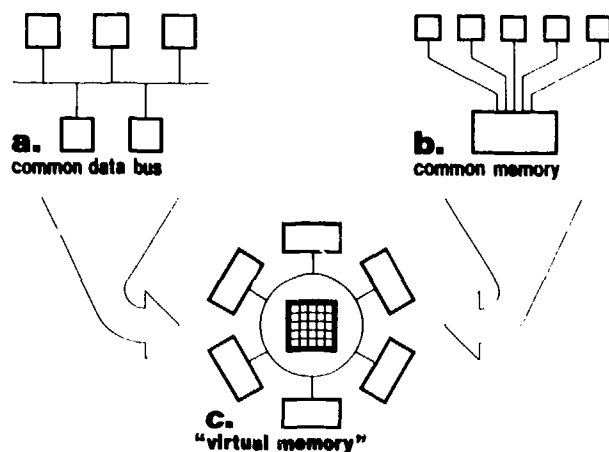


Fig. 2 Evolution of Virtual Memory

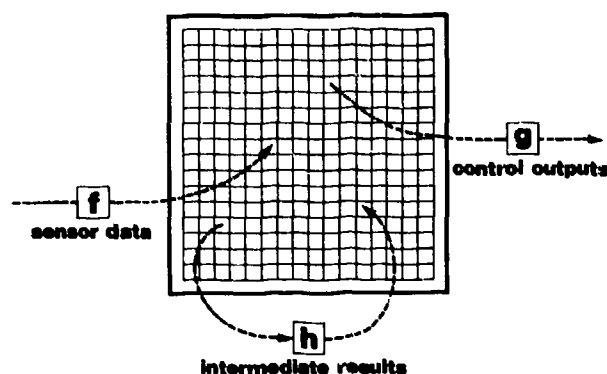


Fig. 3 State Information Matrix

3. VIRTUAL COMMON MEMORY

An architecture which meets all four requirements was developed in the Flight Dynamics Laboratory. It is a combination of the software advantages of the common memory architecture and the hardware advantages of the common bus architecture. The best of these two architectures form the basis of the virtual common memory architecture illustrated in Figure 2.

One of the key advantages of the virtual common memory architecture is that it is a common bus architecture which looks like a common memory architecture to the software programmer. In this architecture each microprocessor simply interacts with a set of information in the virtual common memory that contains all necessary information about the state of the aircraft. This area of the virtual memory is called the state information matrix or SIM.

The SIM is a mathematical abstraction used for organizing all the available information about the state and environment of an aircraft. With this structure all microprocessor functions can be broken down into three sets. The first set of functions takes raw sensor data, the F functions in Figure 3, process, filter, and store it in designated locations within the SIM. Another set of functions, the H functions in Figure 3, take information which is in the SIM, process it, and refine it to produce higher quality data. This could be, for example, a Kalman Filter algorithm. This refined data is stored back in the SIM where it can be accessed by other processors in the system. A third set of processor functions take information from the SIM and processes it for use by the outside world. These are the G functions in Figure 3 and are typically control laws or display algorithms. With the SIM structure, all software programming for each microprocessor has been reduced to a simple set of interactions with the state information matrix.

3.1 Implementation Of Virtual Common Memory

The implementation of the virtual common memory in hardware (shown in Figure 4) utilizes the simple serial bus structure described earlier. Each unit interfaced to the serial bus is referred to as a processing module. A processing module consists of a microprocessor, local memory, transmitter, receiver, and a copy of the state information matrix. Each processing module independently determines which task it must do next. It accesses variables from the local SIM which are needed to do a computation. When the algorithm has been completed, the data and its location in the SIM are placed in the processing module's transmitter buffer. The transmitter circuit automatically searches for an available bus and transmits the information. Every processing module receiver, including the originating processing module, receives the data. Through a direct memory access, the data is then placed in the proper location in the SIM of every processing module. Each processing module maintains an identical copy of the SIM. As far as any processing module is concerned, the SIM appears to be entirely within its own local memory. Using this concept, processors connected by a simple serial bus appear to share one common memory containing all information in the system. This greatly simplifies programming by reducing interprocessor communication to simple reads and writes on a virtual common memory.

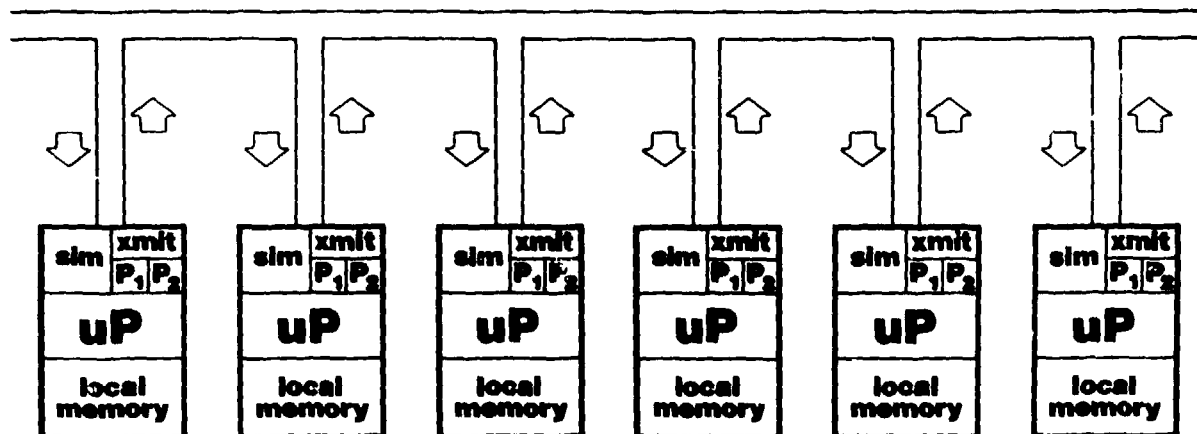


Fig. 4 CRM2FCS Architecture Elements

4. BUS CONTENTION

The virtual memory concept requires a great deal of information transfer and required a new approach to bus contention which would allow the processors to compete for access to a serial bus without the need for a central controller. The bus contention scheme presented greatly increases the efficiency of bus utilization and allows improved bandwidth, expandability, and reliability over other conventional approaches. The technique also permits simple precise scheduling of transmission on the bus to virtually eliminate the effects of transmission delay in the system (Larimer, S.J. and Maher, S.L., MAY, 1981).

Time on the bus is divided into a series of consecutive intervals (slots) that are exactly one transmission word long, 32 to 46 bits, depending on word format. At the beginning of each new slot, all processors with something to transmit compete to fill the slot with a word of data. The resulting massive bus collision is then resolved using a technique called "transparent contention". Transparent contention is a scheme which allows collisions to occur on the bus in a manner such that only one of the colliding messages survives. All other messages are automatically suppressed without wasting a bit of transmission time during the collision. As a result, the slot is filled with one and only one data word and competition moves on to the next available interval.

In order to insure that there is always data available for transmission, each processor maintains a queue of words to be transmitted. As each new piece of data is generated, the processor places it into a first-in-first-out (FIFO) buffer. A special transmitter circuit is then responsible for emptying the FIFO onto the bus by competing for time slots with all other transmitters in the system. This frees the processor from transmission considerations and ensures a constant flow of data onto the bus.

The essential elements of the bus architecture are shown in Figure 6. Three processing modules are shown interconnected by a common serial bus made up of a data line and a clock line. Each processing module consists of an ordinary microcomputer with two I/O devices including a broadcaster (B) and a receiver (R). These devices use the signal on the clock bus to synchronize data transmission and reception. "T" in the figure is a bus termination circuit which generates the clock signal, terminates the clock and data busses, monitors the busses for faults, and generates synchronization pulses for the processing modules.

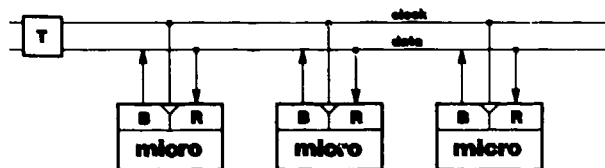


Fig. 5 Essential Architecture Elements

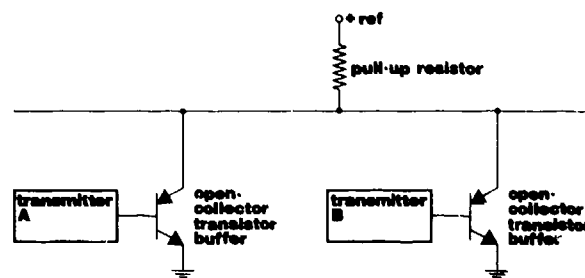


Fig. 6 Transmitter-Bus Interface

Access is granted to the bus on a first-come first-serve basis. While one transmitter is actively using the bus, a logical BUSY signal is maintained which prevents any other transmitter from initiating a broadcast. This eliminates many conflicts, but the probability is high that more than one transmitter will initiate a transmission on the same clock pulse. When this happens, some other method is required to resolve the bus contention problem.

The solution is found by observing exactly what happens when two transmitters try to put data on the bus at the same time. Figure 6 shows each transmitter connected to the bus by an open collector transistor buffer. When the transmitter puts a "0" on the bus, the output transistor drives the bus to ground. To transmit a "1" the transistor is turned off, allowing the bus to float high, because of the pull-up resistor. As long as no transistor is turned on, the bus will remain floating at a logic "1"; but if any of the transistors turn on, the bus will be pulled to the logic "0" state.

The net result is that logic zeroes have an inherent priority on the bus. Because a "1" is transmitted by releasing the bus while a "0" is transmitted by actively pulling the bus low, units transmitting zeroes will always have priority over those sending ones. This fact is used to develop an effective arbitration scheme.

The key to this scheme is that every transmitter constantly compares what it is trying to put on the bus with what is actually there. In the event of a disagreement, the transmitter simply stops sending, waits for the bus to become available again, and retransmits. This approach works because when any two processors disagree, only one of them detects the disagreement and drops off. The other transmitter does not detect the difference, because of the logic level priority, and continues its transmission. No bus time is wasted because one message is completed without interruption.

This concept works equally well for any number of transmitters in contention. If ten transmitters start simultaneously, they all send in parallel until there is a disagreement. Any transmitter attempting to send a one will then drop off while those transmitting zeroes will continue. Eventually, only one transmitter is left and it completes its transmission, completely unaware that it has been contending for the bus.

4.1 The Multi-Bus Concept

The bus structure described represents a very simple way to interconnect a large number of autonomous processors without need of a central controller. However, a single bus system of any kind is generally unacceptable from a reliability standpoint. At the very least, some form of redundancy is required in order to avoid a potential single point

failure in the system. Also, a single serial bus has a finite bandwidth. A large system of processors exchanging massive amounts of data can quickly saturate such a bus, making further system expansion impossible. The approach proposed in this paper is ideally suited to expansion to as many busses as are needed to meet the reliability and throughput requirements of most any system (Larimer, S.J. and Maher, S.M., MAY 1981).

This bus design has tremendous flexibility. There are four serial busses used in the in-house program. The bus bandwidth of the system is exactly four times that of a single bus and can be expanded still further with additional busses. Reliability is also advanced. Selection of an alternate bus in the event of a failure is instantaneous and automatic because processor to bus connections are continuously reconfiguring.

5. TASK ASSIGNMENT RULES

Another major outgrowth of this research has been the development of a method for programming a multiprocessor system (Larimer, S.J. JUNE, 1981). Programming a system consisting of a large number of processors can become a formidable task. Figure 7a shows how four different processors might be programmed in a multiprocessor system. As each processor completes a task it goes on to the next one immediately. This approach is very difficult to synchronize. For example, processor 2 does task A while processor 4 does task B and processor 3 does task C which combines the results of tasks A and B. If task B is not completed before task C is started, then task C will not have the information needed to complete its calculations. This possibility can greatly increase the complexity of the software. A second problem with this programming technique is that it is very difficult to modify. If a block of software requires reworking or a new algorithm must be added, the timing of the software will be changed. Synchronization must be maintained between certain tasks and guaranteeing the synchronization requires revalidation of all software. One small change in the software will therefore influence the software validation of the entire system.

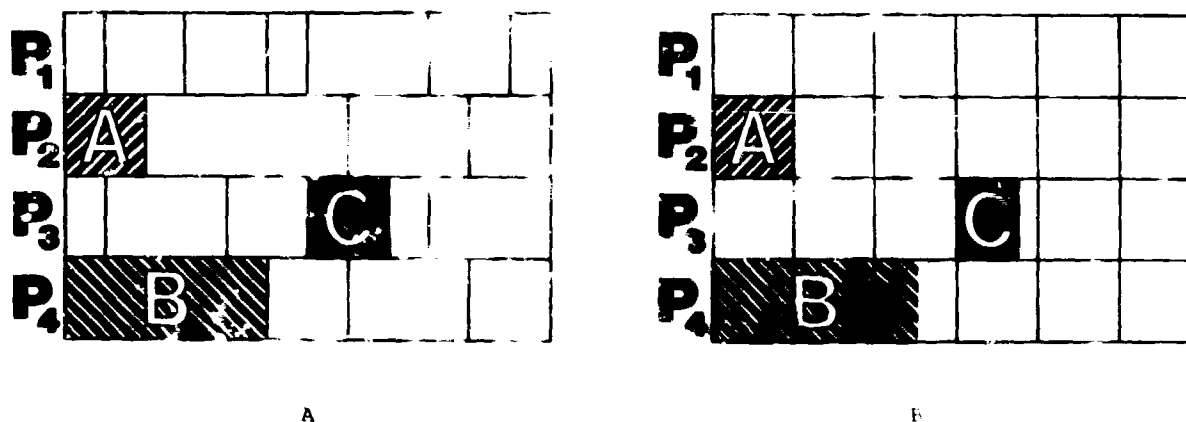


Fig. 7 Sequential vs. Quantized Software

5.1 Quantized Software

The programming method used in the CRM2PCS is called the quantized-software approach (Figure 7b). Every task is given an integer number of time intervals depending upon the length of the task. In this particular system, every interval is one millisecond long and is referred to as a millimodule. Every task is some integer number of millimodules long. For example, a task which normally be executed in 1.5 milliseconds would be allocated two complete millimodules. This allows control over which tasks are being performed during any given interval of time, so strict synchronization of tasks can be maintained. Data is exchanged only on boundaries between millimodules. As a result, the availability of data or subsequent tasks is known during any millimodule. Since software modules are the same size, they can be easily interchanged.

The quantized-software approach obviously sacrifices some throughput, for example a 1.5 millisecond task now takes two milliseconds, and therefore is less efficient than the continuous software method. However, the sacrifice in throughput is well justified in view of the added software simplicity and flexibility. Additional throughput can be added by simply adding more processing modules while maintaining the software simplicity and flexibility.

5.2 Reconfiguration

The reconfiguration rate of the CRM2PCS is once every ten milliseconds. This rate is arbitrary and could be adjusted to a slower rate if data gathered from the laboratory instrumentation indicates the rate is unnecessarily high. Figure 8 shows how reconfiguration fits into the software scheme. A processing module health status table is maintained in the STM. At the beginning of every major frame the status table is "zeroed out", as in task A of Figure 8b.

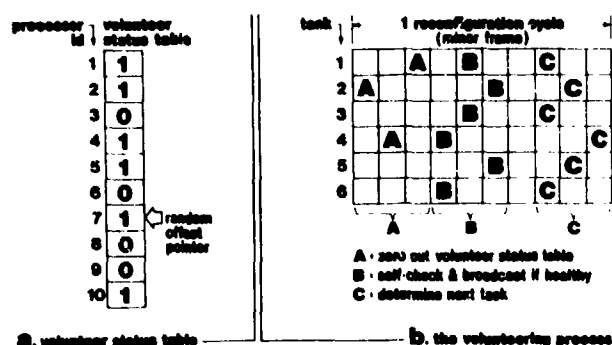


Fig. 8 Volunteering and the Volunteering Status Table

Each processing module must then perform a self-check to determine its own health, task B in Figure 8b. The processing module then broadcasts its health status which becomes available in the SIM processor status table. A processing module can then determine the task set it will be required to do during the next major frame, task C in Figure 8b. To do this, each processing module accesses a specific variable in the SIM. The random nature of the variable is used to generate an offset pointer which every processing module uses to determine its starting point in the SIM processor status table. In Figure 8a, for example, the random offset pointer is pointing at processing module 7. Processing module 7 will therefore do the highest priority task set during the next major frame. Processing modules 8 and 9 have "0" status indicating they are unavailable for task assignment. Processing module ten will determine that it must do the second highest priority task set since 8 and 9 are unavailable. Similarly processing modules 1 and 2 will do task sets 3 and 4 and processing modules 4 and 5 will do task sets 5 and 6. This process is repeated every major frame so that task sets are randomly distributed among functioning processors.

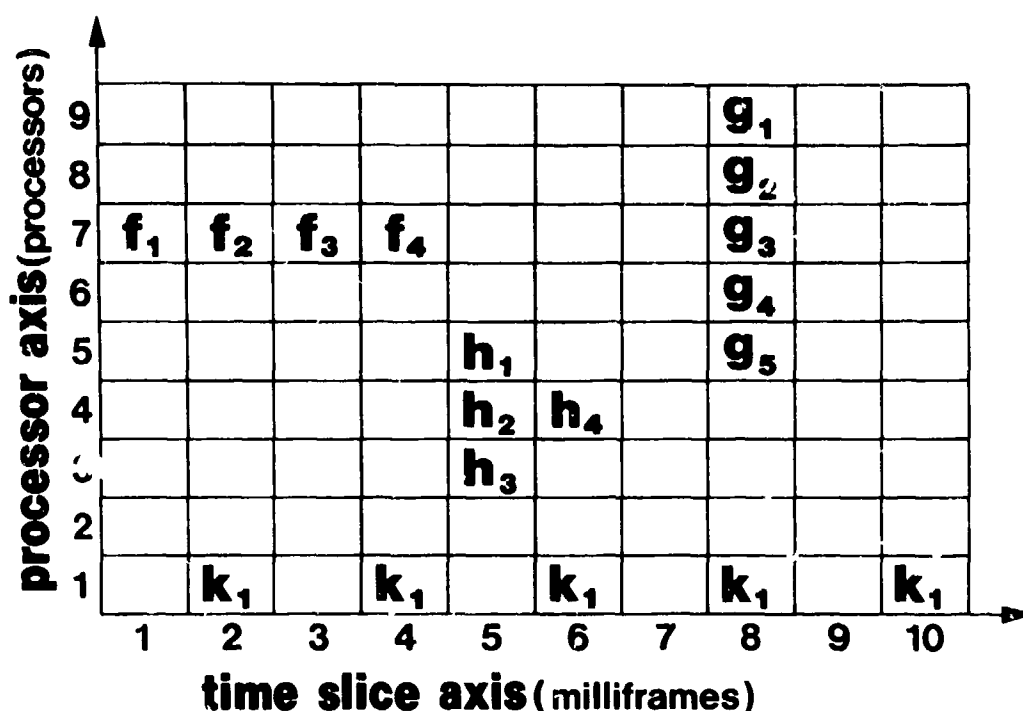


Fig. 9 A Generic Task Assignment Chart

5.3 Task Assignment Chart

The task assignment chart is used to organize all of the one millisecond software modules (millimodules) to be used in the system (Larimer, S.J., May 1981). Figure 9 illustrates how the task assignment chart is organized. The vertical axis represents the number of processors in the system. The horizontal axis is divided into one millimodule time increments (milliframes). Ten milliframes form a minor frame and 3 minor frames complete a major frame. Every task is performed at least once during every major frame. To use the chart, the programmer first divides a function into a group of subfunctions each of which requires at most one millisecond to execute. Each of these subfunctions is then designated as a millimodule and placed in a convenient location in the task assignment chart. In Figure 9, function $F(f_1, f_2, f_3, f_4)$ executes in four consecutive time intervals beginning with milliframe 1. Function $G(g_1, g_2, g_3, g_4, g_5)$ executes entirely in parallel requiring five processors and only one milliframe. Function $H(h_1, h_2, h_3,$

and h4) first generates intermediate results in parallel and then combines them in milliframe 6. Various iteration rates may be achieved by assigning the same function several times in the same chart as shown for function K(k1). The task assignment chart is used to easily distribute tasks among available processing modules.

5.4 Task Assignment Compiler

The task assignment compiler is currently under development at the Flight Dynamics Laboratory. It is an automated method for generating the task assignment chart. The task assignment chart rapidly becomes difficult to work with as the number of processing modules increases and the number and variations in rate of tasks increases. Data concerning each of the millimodules is input to the task assignment compiler and a complete task assignment chart and data file for the processing modules is generated.

Millimodules are given an identification number or name when they are written. The millimodule identification, required repetition rate, and data I/O requirements are input to the task assignment compiler. The compiler automatically rearranges millimodules to make room for new millimodules and indicates to the user whether additional processing modules will be required to accomplish all tasks. This method of simplifying the software development will further reduce the workload for the programmer.

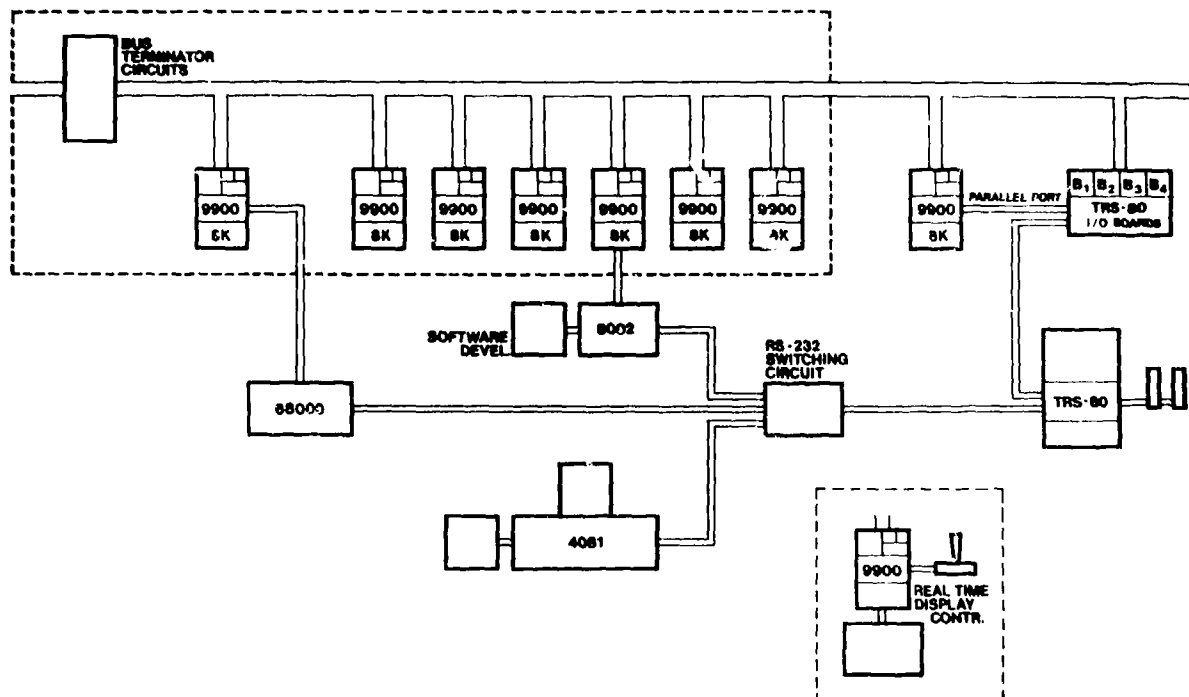
5.5 Software Simulation

A software simulation of the CRM2FCS hardware and software is also being developed at the Flight Dynamics Laboratory. The simulation output will be compared to results obtained from the laboratory system. Discrepancies between the simulation and laboratory system will be analyzed and improvements made to the simulation or laboratory system as required until the simulation can be verified as accurately representing the laboratory system. The software simulation can then be used to predict the effects of changes to the baseline system without having to make changes to the system itself. The effects on throughput or bus utilization of adding more processing modules, using a different microprocessor, or changing the transmitter-receiver hardware can be studied. The software simulation is expected to be a valuable tool for analyzing advanced configurations of the CRM2FCS.

6. LABORATORY IMPLEMENTATION

An effort is under way at the Flight Dynamics Laboratory to demonstrate the CRM2FCS concepts. Data gathered from this in-house program will be used to quantify the extent to which expected benefits and limitations of the architecture are met. A validated software simulation of the system will then be used to project throughput, fault tolerance, and other quantifiable characteristics of modifications to the baseline hardware.

The in-house facility, shown in Figure 10, has been designed to maximize data gathering, data reduction and programmability of the system. The basic CRM2FCS architecture is represented by the six processing modules and bus termination circuit shown in the figure. The remaining blocks represent interfaces to an aircraft simulator, cockpit CRT display, data gathering, data reduction, and software development facilities.



A processing module consists of a 16-bit microcomputer, 8 Kwords of memory, and custom engineered transmitter, receiver, and state information matrix (SIM). At this writing a processing module has been successfully implemented in the laboratory. The custom circuitry uses small and medium scale integrated circuits. A future effort could put the circuitry in a single large scale integrated circuit.

The block labeled "68000" is a state of the art 16-bit microcomputer which will be used for a single axis digital aircraft simulation. It is interfaced through a dedicated processing module to demonstrate one method of accessing external system components such as sensors and actuators. A follow-on effort will use an analog computer to do more complex aircraft simulations.

The block marked "8002" is a Tektronix microprocessor development system. It is used for both hardware and software development. It has a direct hardware interface to a processing module's microprocessor. The "8002" can download software to the processing modules prior to a simulation run. After the simulation the "8002" is used to make software modifications based on data gathered during the simulation. The new software can then be rapidly downloaded and the system brought up for another run.

A Radio Shack TRS-80 is used in conjunction with a dedicated processing module and custom serial bus interface to gather data during a simulation run. The processing module is used to monitor the history of specific variables in the SIM. The serial bus interface is used to gather raw data from each of the four serial busses. The TRS-80 then processes the data to pinpoint specific problems and to determine bus utilization and system throughput. The TRS-80 also controls the RS-232 switching circuit. This circuit allows data and software to be easily transferred between the major components of the test system.

The real-time display controller is a microprocessor-based color graphics display which can be configured as a cockpit instrumentation display or be used to monitor the system status real time. The display controller also has a joy stick input which can be used in more advanced aircraft simulations. The real time display controller also demonstrates the ease with which the architecture can be interfaced to other aircraft subsystems.

The Tektronix 4081 is a stand alone minicomputer with graphics capability and a link to a main frame computer. It is used for further data reduction and display and for the development of complex software for the millimodule compiler and software simulation.

7. CONCLUSION

There are three major potential benefits to designing a flight control system using the methods described in this paper. The first is simply expandability as system needs grow. It is a well known fact that from the time the first model of a particular aircraft rolls off the assembly line until the last one lines up in mothballs, there are innumerable changes that occur to the system. This causes excessive increases in cost due to the difficulties of changing hardware and adding new software to the system. The CRM2FCS approach has the potential to greatly reduce these costs. Modularity of both hardware and software allows considerably easier expandability.

A second potential benefit is the ability to reduce software costs which are the single biggest cost in digital systems today. By designing an architecture that is inherently easier to program, the cost of programming, maintaining, and updating software can be greatly reduced. This contributes to a reduction in life cycle costs.

The third potential benefit is the possibility of greatly reducing unscheduled maintenance. With the present redundant flight control computers, if any component of the computer has failed the aircraft is not allowed to take off. As digital technology progresses, it will become practical to configure the CRM2FCS with as many as one hundred processors. If only 40 processors are required to accomplish the necessary processing there will be 60 spare processors. A requirement that 20 spares be available before the aircraft takes off leaves 40 processors that can fail before the aircraft is grounded. When scheduled maintenance occurs, any failed processors can be replaced. Since it is unlikely 40 processors will fail between maintenance periods, the goal of no unscheduled maintenance can be closely approached.

REFERENCES

Larimer, S. J., June 1981, "Managing Software in a Continuously Reconfiguring Multi-Microprocessor System", Proceedings of the 1981 Joint Automatic Control Conference.

Larimer, S.J. and Maher, S.L., May 1981, "A Solution to Bus Contention in a System of Autonomous Microprocessors", Proceedings of the IEEE 1981 NAECON Symposium.

EXPERIENCES WITH THE EXPERIMENTAL

FFM - MCS

Hermann v. Issendorff

FGAN - FFM
Königstr. 2
5307 Wachtberg-Werthhoven
Germany

SUMMARY

The FFM-Multicomputersystem was built up to investigate the utilization of microprocessor based computing networks for the various requirements of embedded data processing and control in military systems. Being designed as an adaptable building block system the FFM-MCS is serving as a testbed for research on distributed data processing. The paper deals with a general method for the design of process-networks followed by the adaption of adequate hardware-networks. Several types of messages are introduced for efficient and safe communication between autonomous process-modules. Finally some improvements of the hardware building block system are presented.

1. INTRODUCTION

Distributed systems seem to be particularly well suited for embedded data processing and control in military applications like airborne systems. Apparently there are numerous advantages which make distributed systems preferable to conventional monolithic systems.

Some attributes seem to be of particular importance to airborne systems:

- . Distributed systems can be designed as a network of autonomous computers. A network of this kind constitutes a good base for the construction of fault tolerant, fail soft and damage resistant architectures.
- . Distributed systems can be built up with only a few types of different components which may even be massproduced commercial products. This may result in easier maintenance and repair as well as in lower hardware costs.
- . Last not least there are indications that software production, i.e. programming and testing could become much easier than with conventional data processing systems. The same holds for subsequent extensions and changes of the network. Hence there is a good chance that life-costs of distributed systems can become considerably lower.

On the other side the science of distributed systems is still in an infant state. Even the term distributed system is not rigidly defined yet. There is no profound knowledge how to control a network of autonomous nodes. The data transfer in a distributed system of this kind will be greatly increased compared to monolithic systems. It is not clear so far if this problem can be sufficiently solved or if it presents a severe restriction to the utilization of distributed systems. There isn't any design methodology available and there exists no language which supports programming of autonomous processes and their communication. The promising aspects on the one side and the unsolved or even undetected problems on the other side gave rise to a long term research project at the FFM in Werthhoven. Some of the main results of this work are presented in this paper.

It begins with a description of the general approach to decompose the data processing task of a given application, the formation of a network of process modules and the adaption of a hardware network to the network of process modules. Paragraph 3 describes the hardware building block approach and the experimental FFM-MCS (Multicomputersystem). Paragraph 4 deals with the important subject of communication especially with the definition of messages and simultaneous message execution. Improvements of the hardware building block system which increase flexibility and decentralization will finally be discussed in paragraph 5.

2. THE DISTRIBUTED SYSTEM DESIGN APPROACH

The first step towards a distributed system consists of a decomposition of a data processing task into a set of functions which are interrelated by their input and output parameters. This first step resembles very much that of Mascot (Mascot Suppliers Ass., 1980). But while our functions are nearly identical with the activities of Mascot, we do not introduce IDA's (Intercommunication data areas). A function is not a welldefined object. It is likely that this object has a minimal interface in relation to its complexity but that may not be true. A function may be further partitioned, as vice versa adjacent functions may be combined to one function. The final size of a function will be confined later by the features of the hardware network. An example of decomposition is shown in figure 1. Simple as it is it shows already how a processing task can be executed by macropipelining to increase the throughput (Händler, W., 1973). In general a data processing task will be decomposed in many more functions resulting in a more complex network. Functions in separate paths are independent and hence may be executed in parallel.

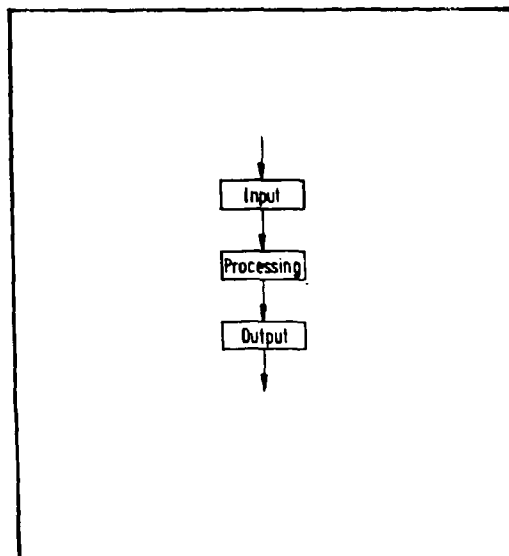


Figure 1: Functional decomposition

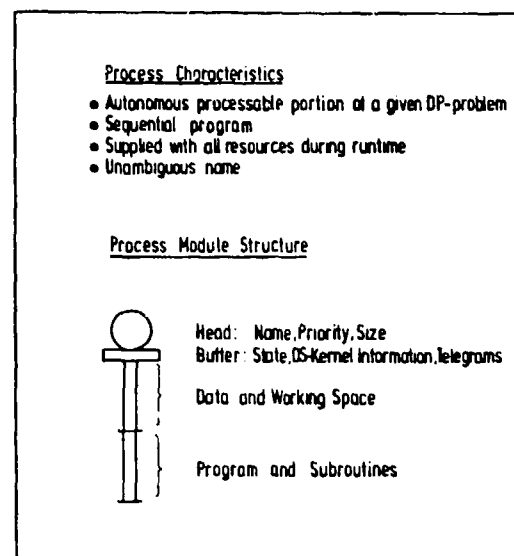


Figure 2: Characteristics and structure of a process

In a next step each function is realized as a sequential program and embedded in a process module. Any conventional higher order language suitable for the kind of application may be used for the internal programming of a process module as long as the special characteristics of the process module are taken into consideration. A process module is selfcontained and autonomous. Figure 2 shows the process characteristics and the module structure. The only way how a process module can be accessed is by messages from other modules. Vice versa a process module cannot access anything else

but another process module. Hence any data base or a device must be embedded in a process module. Such a process module is called a monitor. All functions of the operating system are represented as monitors too. Monitors are not as mobile as ordinary process modules are. Figure 3 represents an example of a process-network.

Communication between process modules takes place by sending or receiving messages (Walden, D.C., 1972). The sending and the receiving process must agree to the message before the transfer is actually executed. This message concept makes the processes really autonomous and protects them against erroneous information from other processes. Communication by messages will be treated in detail later on. But it may be mentioned that the message type being introduced in the language Ada is not sufficient and has to be backed up by other types of messages.

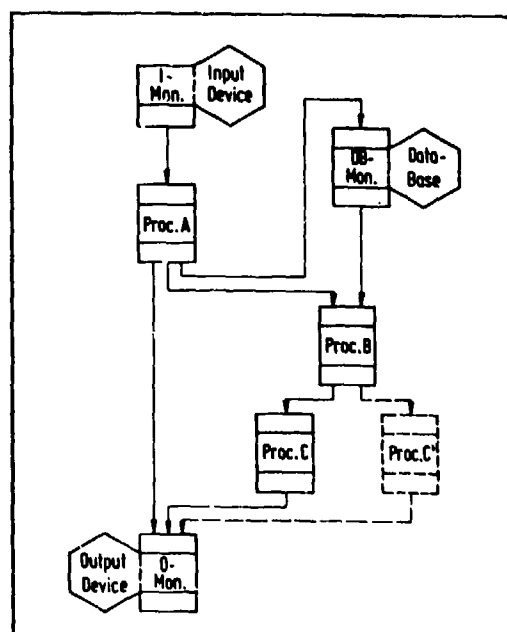


Figure 3: A typical process-network

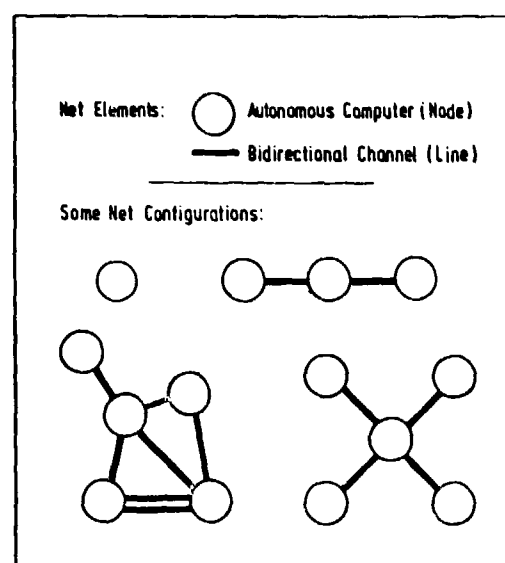


Figure 4: Building block system I

What has been constructed up to this point is a pure software-network. There is nothing said so far about the hardware system into which the software-network is to be loaded and where it is to be processed. Indeed the software-network could be loaded into any computer system no matter if it is a single processor architecture, a multiprocessor architecture or even a computer-network. This independence of the hardware architecture represents an ideal base for reconfiguration. On the other hand it allows to adapt the hardware architecture to the requirements of a given application. This could be particularly beneficial if the hardware would be composed from a building block system consisting of a few and highly standardized components.

3. THE EXPERIMENTAL FFM-MCS

A building block system with a high degree of flexibility on which our research is based has been described earlier (v. Issendorff, H. and Grünwald, W., 1980). Figure 4 and 5 depict the main features. The hardware-network can be adapted to a given process-network in two levels. At the higher level autonomous computers (nodes) can be interconnected by channels (lines) to an arbitrary network, i.e. to a network with an arbitrary number of nodes which are interconnected in an arbitrary manner. The channels are only necessary in a logical sense. Several channels can always be combined to one bus if this is desirable. No other difficulties would arise from this but possibly a bus contention problem.

At the lower level each autonomous computer can be equipped with several processors, memory modules and peripheral controllers besides of the communication links which serve for the interconnection of the channels.

In general a process-network will not have an equal flow of information between processes. There will be groups of processes which communicate heavily while others will have a rare or small data transfer only. For efficiency reasons these groups are preferably clustered in one node or at least in nodes which are directly connected. A node which contains several processes should of course be equipped with several processors.

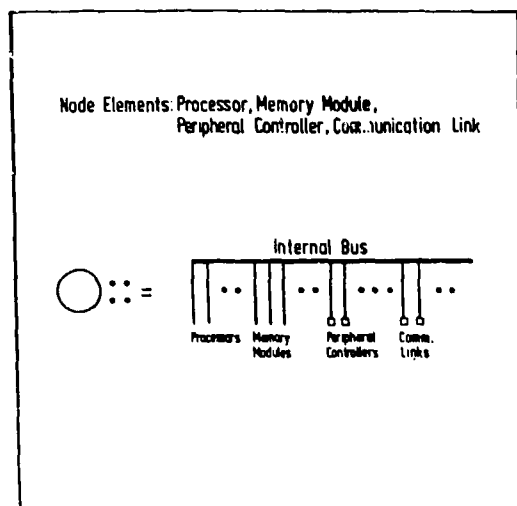


Figure 5: Hardware building block system II

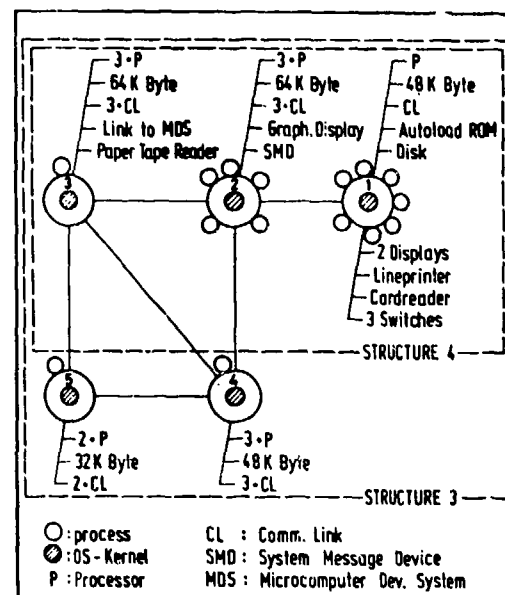


Figure 6: Some configurations of the FFM-MCS

The experimental FFM-MCS is constructed in very much the same way. Some restrictions and some compromises had to be accepted because of the utilization of commercial products. Base of the building block systems is the SUE-minicomputer from Lockheed Electronics which has a bus-oriented architecture and can be extended up to 4 processors. Figure 6 exhibits two different configurations. Each node has its own operating system kernel. This kernel handles several types of supervisor calls, e.g. calls to allocate processors to processes which are ready to be started and calls to control the communication with adjacent nodes. The set of operating system kernels represent the basic operating system of the network. Figure 7 depicts the logical system structure which displays the different layers and the kind of communication between them.

Bootstrapping of the MCS is done stepwise. It begins with a node which is reset by hand. This node then resets, loads and starts his neighbour nodes, which then do the same with their next neighbours until the whole system is bootstrapped. It does not matter where the bootstrap begins provided that the first node has access to a data base where the basic operating system is stored. But each bootstrapped path must be predefined in order to guarantee that each node will be bootstrapped only once.

4. COMMUNICATION

A conventional monolithic computer permits direct access to (globally defined) data from any point of a program. Hence data do not have to be moved very often. This architecture offers high speed performance but suffers from a rather low reliability because one single fault may destroy the whole system. A distributed system instead permits high system protection on the expense of a high load of data transfer. The control of the data transfer between autonomous process modules even increases this load. Communication is therefore the most important topic of distributed systems and has carefully to be investigated in order to preserve system efficiency. The results which we got in this area are presented in the sequel.

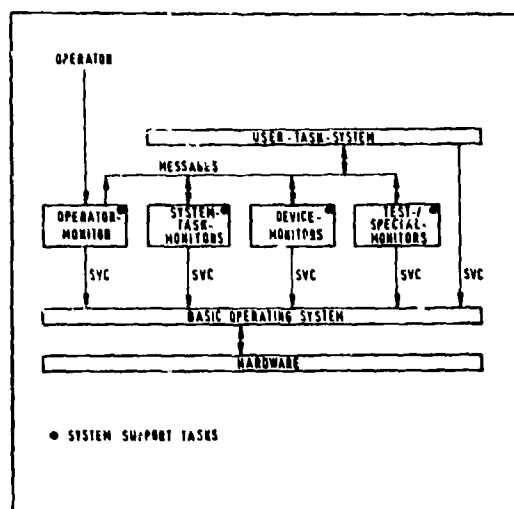


Figure 7: Logical System Structure

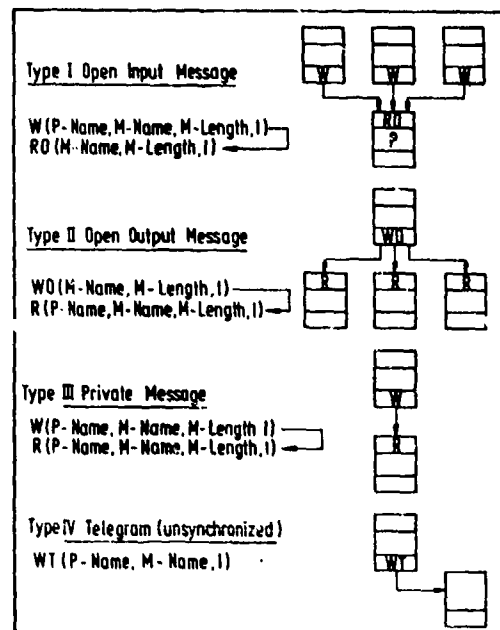


Figure 8: Types of Messages

Several types of messages are needed for practical reasons like efficiency or safety. They are listed in figure 8. Each message consists of a write-instruction in one process and a corresponding read-instruction in another process. The parameters in the brackets of both instructions will be checked prior to the information transfer. The first type is called an open input message. This type corresponds to the rendezvous concept introduced in Ada (Ichbiah, J.D., et al., 1979). An open input message is necessary if there is a receiving process which has to accept messages from several other processes. The receiving process is waiting until another process contacts him by transmitting his name and where he is located. This results in a most extensive protocol of 4 steps (figure 9). The next message type called open output message is complementary to the first type. This message takes care of sending data to any process which applies for them. The sending process does not know the receiving process until he gets a call which tells him name and location of the receiver. The protocol consists of 3 steps only. A third message type is called private message. This type has been used in CSP (Hoare, C.A.R., 1978). Here both the sending and the receiving process know each other by name and location. This message permits a maximum of protection against unauthorized access by other processes. The protocol consists of 3 steps as the open output message.

The open input message is mainly used to transfer information to monitors, e.g. to a printer-monitor. The open output message on the other hand serves for the case where several processes compete for one message which is repeatedly produced by the sending

process. This arises for example if a process is duplicated for speed or fault tolerance reasons as indicated by dashed lines in figure 3.

The open output message is introduced for efficiency reasons only. It could be substituted by an open input message followed by a private message with an expense of 7 steps altogether. In this case the open input message serves merely to transmit name and location of the calling process. Even the private message could be substituted by the public input message yet with the disadvantages of a longer protocol and a reduced protection.

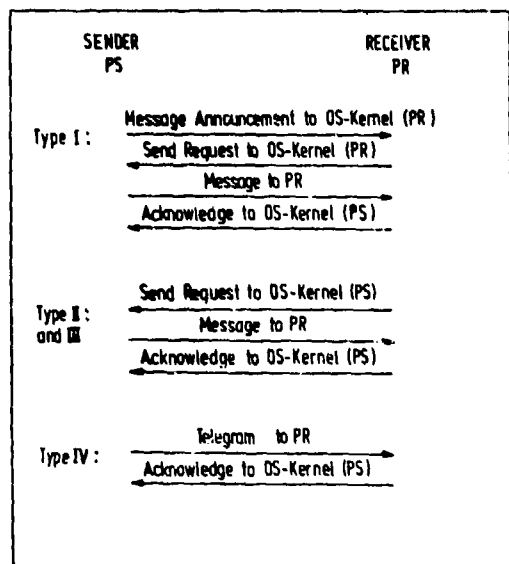


Figure 9: Message Transfer Protocols

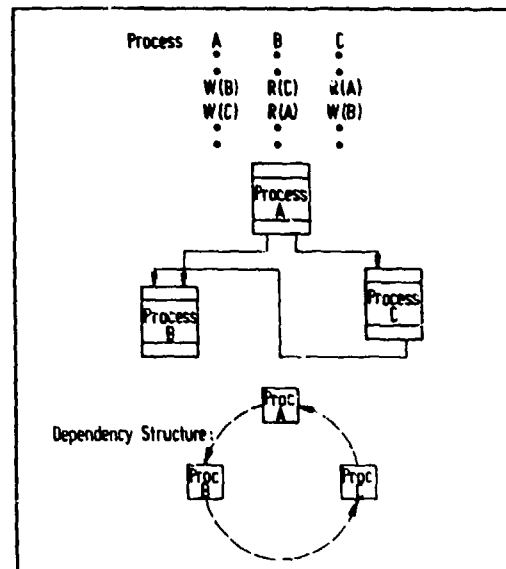


Figure 10: A Simple Case of a Deadlock-Ring

Message type IV is called telegram. It permits transmission of information without the control of the receiving process. Only one word at a time can be transferred. This message type is very useful if information is to be transferred which is not time critical. For example such information could be either slowly changing data from an input device or status reports or the like. A telegram has a two step protocol.

There is another important subject with regard to communication which has to be discussed too. By inspecting an average process module it will be recognized that there are a group of several message-instructions at the beginning and another group at the end of the module with only some of them scattered in between. The instructions at the beginning will mostly be read-instructions, collecting input parameters from other processes while those at the end will mostly be write-instructions which distribute the results to other processes. There seems to be no reason why the read- and write-instructions should not be executed in the same sequential order as all other instructions in a process module. And indeed no problem will show up as long as there are only a few processes with little communication between them. But this changes with an increasing number of processes especially if they are closely interrelated by messages.

A first problem will be that the software-network becomes trapped in deadlocks though it may be logically correct. The effect is explained in figure 10. The messages in each of the processes A, B and C are assumed to be independent with regard to their contents. The message cannot be executed however because of the order of the read- and write- instructions. Each process tries to execute the instruction of a different message and is waiting for signals from another process, therefore. This results in a deadlock-ring which is being displayed in a dependency structure.

A deadlock-ring can easily be broken up by reversing the order of messages in one of the processes. Moreover, they are easily to be detected. Checking for deadlock-rings could even be done during compile-time if there were a suitable higher order language for distributed systems.

But there is another problem which arises with the sequential execution of message-instructions: The average delay time of a message increases with the number of all messages in the process-network. The overall delay time becomes proportional to the number of processes if the processes are interconnected to a ring (figure 11).

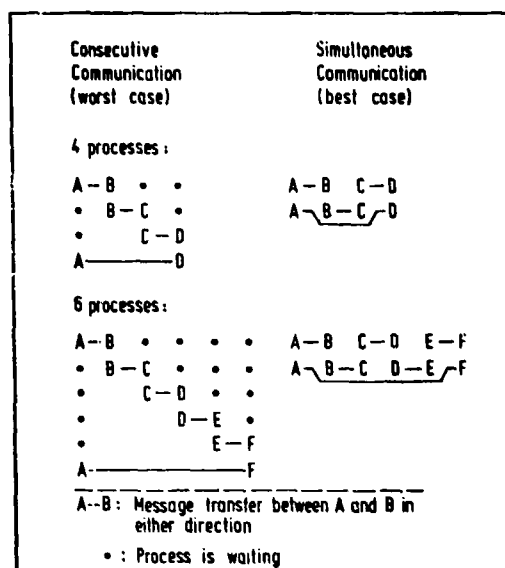


Figure 11: Consecutive and Simultaneous Communication

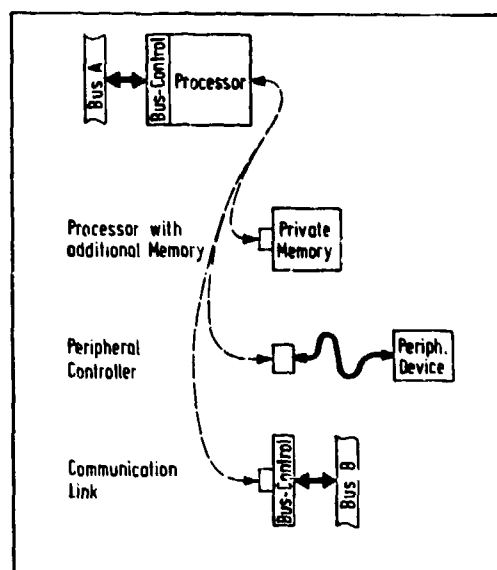


Figure 12: The Janus Processor

Both problems, the deadlock problem and the delay time problem would be solved if instead of executing the message-instructions sequentially this could be done simultaneously. This would reduce the total time for communication in the example of figure 11 to 2 steps. Simultaneous execution is possible only for messages which are not related with regard to their information contents. This means that the contents of one message must not be a function of another. This restriction could not be easily implemented but there is another stronger restriction which is clear and simple. It holds under the additional assumption that there will be a separate buffer for each message-instruction: Blocks of messages i.e. sequences of message-instructions which do not contain any data processing may be executed simultaneously. Simultaneous execution means that the block of message-instruction will be repeatedly run through until all messages are carried out.

5. IMPROVEMENTS OF THE HARDWARE BUILDING BLOCK SYSTEM

The FFM-MCS is operational for about two years and has been used for several test applications. Detailed measurements have been carried out to localize bottlenecks and to get a better insight into the dynamic behavior of the system. (Neumann, G., et al., 1980). The evaluation of the results led to several improvements with regard to the hardware structure. These improvements are currently being implemented in a new experimental system which will be used for further research on reliable, fault tolerant, fail soft and damage resistant systems. The name of the new system is MICON (Micro-computer-network).

Key component of this building block system will be a microprogrammable microprocessor with two identical input/output ports as his main feature (figure 12). Because he can look and act to two sides at the same time he is called Janus-processor. The second port can be used in three different ways. It can be used for the connection of private memory which doubles the address space of the processor. But even more important it allows to store code and data which are heavily accessed and reduces the nodal bus contention thereby.

With a peripheral device connected to the second port, the Janus-processor would serve as an intelligent device controller and could take care of the appropriate device-monitor at the same time.

The Janus-processor finds the most important application as an active communication link, i.e. with the second port being directly or indirectly connected to the bus of another node. The Janus-processor is able to handle the message transfer between the two nodes all alone, a work which has to be controlled in the MCS by a complex and lengthy dialogue between the masterprocessors of both nodes. While in the MCS the transfer of a message to an adjacent node takes 4.5 ms plus additional 33 μ s/word, a transfer in MICON will probably be reduced to something like 250 μ s plus 10 μ s/word. (Average instruction execution time in both systems is about 3 μ s.)

A new design of the node internal bus will be another major hardware improvement. The nodal bus arbitration will be piecewise attached to all processors and memory modules which are plugged to the bus and therefore be totally decentralized. The number of processors which can be plugged to the bus is merely restricted by the number of open slots and may be as high as 16.

The changes of the bus control permit to distribute the functions of the master-processor of the MCS to all processors of each node. The contention problem of the master-processor has such been eliminated, too.

Acknowledgement: This research would not have been possible without the cooperation of many coworkers. Besides of W. Grünwald and G. Neumann who have been mentioned already before, I would very much like to thank W. Jansen who is taking care of the hardware, a contribution which cannot be valued highly enough.

References

- Händler, W., 1973, "The concept of Macro-Pipelining with high availability"
 El. Rechenanl. 15, Nr. 6, pp.269-274
- Hoare, C.A.R., 1978, "Communicating Sequential Processes"
 CACM 21, Nr. 8, pp. 666-677
- Ichbiah, J.D. et al., 1979, "Rationale for the Design of the ADA Programming
 Language" SIGPLAN NOTICES, Vol. 14, Nr. 6, 11.4.2
- v. Issendorff H., Grünwald, W., 1980, "An adaptable Network for Functional Distributed
 Systems" Conf.Proc. 7. Symp. on Comp.Arch., IEEE Cat. Nr. 80CH 1494-4c pp. 196-201
- MASCOT, 1980, "The Official Handbook of Mascot" Mascot Suppl.Ass., RSRE, UK.
- Neumann, G., Ackermann, R. u. Grünwald, W., 1981, "Messungen zum Kommunikationsaufwand
 für Prozesse in einem lokalen Rechnernetz" Ber.z. German Chapter of the ACM, Bd.7
- Walden, D.C., 1972, "A System for Interprocess Communication in a Resource Sharing Com-
 puter Network" CACM 15, Nr. 4, pp. 221-230

DISCUSSIONS

SESSION II

REFERENCE NO. OF PAPER: II-6

DISCUSSOR'S NAME: Jim McCuen, Hughes Aircraft

AUTHOR'S NAME: K. Shin

COMMENT: What is the criteria for the need for separate data and control buses? Can the buses operating at 50 megabits employ contention-type protocol?

AUTHOR'S REPLY: 1) To increase data bus bandwidth or to reduce bus contention. If you don't separate them, all control signals (information) should be passed via data bus to processors.

2) No, not as of now. Presently we are considering the MIL STD 1553 serial bus that has a maximum 1-megabit/sec bandwidth. But, it may be feasible when the fiber optics become available for practical use. Note that if you don't separate control bus from data bus, the 1553 bus will not have the 1-megabit bandwidth for data passing.

REFERENCE NO. OF PAPER: II-6

DISCUSSOR'S NAME: Dr. van Keuk, AVP member

AUTHOR'S NAME: K. G. Shin

COMMENT: How do you estimate the importance of an atomic function? I feel this can be a difficult job in a complex system with a low degree of redundancy.

AUTHOR'S REPLY: Practically it is not too difficult although the decision may be to some extent subjective. For example, it is reasonable to give more importance to an atom function associated with flight control than to that associated with navigation.

REFERENCE NO. OF PAPER: II-6

DISCUSSOR'S NAME: G. Scotti, Selenia, Italy

AUTHOR'S NAME: K. G. Shin

COMMENT: You have shown some graphs for Bus Request Profile and Average Bus Access Delay. On what assumption did you define the graphs and did you have confirmation of the correctness of the assumptions via simulation or practical measurements?

AUTHOR'S REPLY: Numbers shown in the graph don't mean anything significant. We assumed there are 20 bus requests over 20 time frames. This could be very bursty, i.e. all 20 requests during the first frame and none thereafter; or uniform distribution during the first two time frames, i.e., 10 requests for each of the first two time frames, and none thereafter--or one request for each of 20 time frames, etc. This is an arbitrary example which has a reasonable sense. Of course, bus access delay can be computed for any bus request profile; therefore the graph in the paper has to be understood as a simple but sensible hypothetical example. Of course, this graph is not obtained from real measurements and does not have to be validated with such measurements since bus access delay can be estimated. Any bus request profile which will be process (or task)-dependent and a random variable.

REFERENCE NO. OF PAPER: II-7

DISCUSSOR'S NAME: H. Timmers, AVP member

AUTHOR'S NAME: S. Wright

COMMENT: Can you give some technical details about the microprocessor you are using?

AUTHOR'S REPLY: The main processor characteristics are outlined in the paper, its important characteristics for our application are its small size and high speed (more than 2000K OPS per second). There have been problems in achieving the correct instruction operation, but Intel has promised a corrected version of the device for this August.

REFERENCE NO. OF PAPER: II-8

DISCUSSOR'S NAME: Dr. von Issendorff

AUTHOR'S NAME: K. Branmer

COMMENT: Would it not be important to include the question of vulnerability into the considerations of which communication structure should be preferred?

AUTHOR'S REPLY: Yes, reduction of vulnerability is one of the major reasons for locally distributed processing and must be considered with respect to the interconnecting network, too. Consider, for instance, a single cut of a transmission line--in the case of the multiple access bus, you would lose all communication; in the case of the layered star, it can mean the loss of a group of equipments; in the case of the star, one unit is cut off; and in the all-to-all network, you lose only one two-party line.

On the other hand, by comparing the cabling cost (see fig. 6) one can easily afford at least a double, redundant bus, and that is usually done. Of course, the two cables should run along different tracks separated as much as possible.

There are several subsequent papers dealing with this problem in more detail.

REFERENCE NO. OF PAPER: II-8

DISCUSSOR'S NAME: H. Timmers, AVP member

AUTHOR'S NAME: K. Brammer

COMMENT: Can you compare the relative benefits of the ARINC 429 bussing concept with the MIL STD 1553B bus?

AUTHOR'S REPLY: In the context of this paper the basic difference is that the ARINC Standard allows only one single source to speak on a given line, which is unidirectional (simplex), while the MIL Standard allows multiple sources to use one bidirectional (partial duplex) line on a time division basis. So the ARINC Standard requires a separate cable for each unit that has information to transmit, whereas with the MIL Standard all transmitting units share the same cable.

Both Standards use a single twisted and shielded pair of wires for the channel line and in both cases the messages are broadcast to all listeners (multiple sink). The bit rate of the MIL Standard is ten times higher than that of the ARINC Standard, but due to time sharing and control overhead of the former, the average data capacity allocatable to the message sources connected to a MIL bus may easily drop below the value possible with ARINC. The absence of bus access management in the ARINC concept facilitates system specification and integration, but ARINC requires definitely more cabling (in general, n times as much as MIL if there are n sources). Also, while the number of transmitter ports is equal for both standards, the number of receiver ports is much higher for ARINC: an equipment listening to x other equipments needs only a single receiver port with MIL, but x receiver ports with ARINC.

REFERENCE NO. OF PAPER: II-8

DISCUSSOR'S NAME: Jim McCuen, Hughes

AUTHOR'S NAME: K. Brammer

COMMENT: I question the future use of ARINC 429 multiplexing due to the increased number of buses required on the latest commercial aircraft, e.g., the 767 aircraft requires over 130 buses and one avionics black box (LRU) requires 22 ARINC 429 receivers. The Airbus A310 provides another example of how ARINC 429 is outdated.

AUTHOR'S REPLY: I understand that this is not a question but a comment. It illustrates some of the points in the paper, thank you.

REFERENCE NO. OF PAPER: II-9

DISCUSSOR'S NAME: P. A. Bross, ESG

AUTHOR'S NAME: S. Maner

COMMENT: Why do you synchronize software for multiprocessors instead of using mailboxes, where processors can access data asynchronously? Why did you use a serial bus for communication between the processors instead of a parallel bus?

AUTHOR'S REPLY: (1) The synchronization and quantization of the software is desired in this architecture because it shows potential for simplifying development, validation, and verification of software in a distributed system. The "mailbox" approach to data access is not compatible with the continuous reconfiguration concept. The state information matrix, however, might be considered a "mailbox" where the "mail" is delivered immediately instead of having to go to the post office to get it.

(2) Although a parallel bus has the potential for being much faster than a serial bus, there are several reasons why we did not use a parallel bus. First, the number of interconnected wires would be very large and would greatly inhibit the degree to which the processing modules could be physically distributed. Also, a failure in a single wire of the parallel bus would essentially cause the entire bus to fail.

REFERENCE NO. OF PAPER: II-9

DISCUSSOR'S NAME: Alan Stern, Boeing Military

AUTHOR'S NAME: S. Maher

COMMENT: Questions concerning failure modes: What is to prevent an "autonomous controller" to seize control of the bus due to a failure of one of the microprocessors? What prevents one microprocessor from writing bad data into all the SIM, adversely affecting flight safety?

AUTHOR'S REPLY: (1) Autonomous control, as implemented in this architecture, has no influence over the actual transmission or reception of information. The transmitters and receivers are independent pieces of hardwired digital logic, designed so that a transmitter can control only one bus at a time. The microprocessor itself has no control over the transmission of data beyond supplying the data to the transmitter input buffer.

(2) We have several methods of "filtering out" faults from the system. These include self-test, hardware voting, watchdog timer, software voting, and "blackballing by peers" where a consensus of the other processors in the system can eliminate a faulty processor. Another method of "fault filtering" which shows great promise for supplying broad coverage and reducing overhead software is the self-checking microprocessor pair (SCMP) implemented by Honeywell in a parallel effort to the in-house program. The SCMP is simply a pair of tightly synchronized microprocessors configured as a single processor. The outputs of each processor are compared bit-by-bit to detect faults. Any of the "fault-filter" methods will offer a certain degree of coverage for the possible variations of a certain type of fault, in this case a processor attempting to fill the state information matrix (SIM) with erroneous information. Hopefully, the total fault filter will closely approach 100-percent coverage for all possible faults.

REFERENCE NO. OF PAPER: II-9

DISCUSSOR'S NAME: R. W. MacPherson, D.N.D., Canada

AUTHOR'S NAME: S. Maher

COMMENT: Your system is highly redundant except for the clock. What happens if it fails? Do you have "reconfigurable clocks"?

AUTHOR'S REPLY: The clock is redundant and is generated by bus termination circuits. For example, a system with four buses, as we are implementing at the Flight Dynamics Laboratory, has four bus termination circuits. Each bus termination circuit has four functions. First, simply to terminate a data and associated clock bus for noise suppression. Second, to monitor both the clock and data bus and eliminate the bus in the event a failure is detected. Third, to generate a 1-MHz clock to synchronize data transmission between processing modules. Finally, the bus termination circuit generates a synchronization pulse every millisecond to synchronize the processing modules at the "millimodules" boundary. Each processing module has a voting circuit requiring at least two synchronization pulses be present simultaneously before accepting the pulse. The bus termination circuits also synchronize the synchronization pulses through a similar voting circuit.

REFERENCE NO. OF PAPER: II-9

DISCUSSOR'S NAME: B. Zempolich, USN

AUTHOR'S NAME: S. Maher

COMMENT: We have had problems with regard to where does fault-tolerant design begin and end. For example, do you include the power supplies in your fault-tolerant conceptual design? Do you consider the bounding of your fault-tolerant design to no single-point failures?

AUTHOR'S REPLY: The architecture described in this paper was intended to be a research effort aimed at implementing a flight control computer using distributed processing techniques. The resources were not available to study any larger segment of the flight control system, nor would it have been appropriate to do so. Presently, there is much work being done in this area. Once we have completed trade-off studies we should be able to understand the advantages and disadvantages to the many possible methods of implementing a distributed fault-tolerant computer complex, we can then expand our efforts to include larger segments of the system such as power supply, sensors, actuators, etc.

REFERENCE NO. OF PAPER: II-10

DISCUSSOR'S NAME: J. T. Martin, Ferranti

AUTHOR'S NAME: H. von Issendorff

COMMENT: The system was likened to MASCOT. A program running under a Mascot Kernal can be slowed by real time interrupts. How does the system cope with real time interrupts and what effect do they have?

AUTHOR'S REPLY: There are no interrupts in our system. Each event coming in from the outside is received by a monitor which then may inform other processes by sending messages.

REFERENCE NO. OF PAPER: II-10

DISCUSSOR'S NAME: K. Shin, Rensselaer Polytechnic Institute

AUTHOR'S NAME: H. von Issendorf

COMMENT: Simultaneous message communication is obviously superior to sequential one. However, there must be a way of handling precedence constraints existing in process communication which may force messages to be dealt with sequentially. This is needed even for the case when the message passing is the communication method.

AUTHOR'S REPLY: As pointed out in the presentation already, simultaneous communication is not allowed if the content of the messages depend on each other. The precedence constraint in our system is that only blocks of messages with no data processing in between may be treated simultaneously. This restriction is sufficient because each message has its own private buffer.

SAVANT - A DATABASE MANIPULATION TECHNIQUE FOR SYSTEM ARCHITECTURE DESIGN VERIFICATION AND ANALYSIS

by

Dr A. A. Callaway

Flight Systems Department
Royal Aircraft Establishment
Farnborough, Hampshire
England

SUMMARY

SAVANT - System Architecture Verification and Analysis Technique - is a computer program developed within RAE specifically to provide a tool for automatic system design verification and analysis. Its application is oriented towards loosely-coupled bus connected systems but is not exclusively confined to these. Flexibility has been built into the program to characterise aspects of the system architecture. The SAVANT program provides the facilities for interactively initiating, extending, modifying, filing and retrieving the database, which represents various facets of the system under investigation, and for configuring a system from the database information. The system thus configured can be analysed in a number of ways and the analyses performed can suggest how the basic information should be modified in order to correct errors and inconsistencies or to improve efficiency, and so on. A consistent system can be further modified and tuned, although SAVANT still checks the validity of all operations performed. Finally, the user is able to 'firm up' the system when it has reached a satisfactory state, producing design requirements and a system description in a form which can be input as a schedule to a bus control processor.

1 INTRODUCTION

The configuration of avionic systems has seen a move in recent years away from the concept of a network connected system controlled by a large central processor towards a more federated type of architecture, with digital processing embedded in various subsystems and with the majority of system data communicated by means of a multiplex data bus. A number of recent papers have justified this approach (1, 2, 3), and the purpose of this paper is to introduce a software technique to assist in the design of such systems.

The type of data bus which has become accepted for avionic system use is that known as Mil Std 1553B (ref 4) in USA and Def Stan 00/18 (Part 2) (ref 5) in UK. This is a 1 Mbit/s command-response serial bus where the system data traffic is under the software control of a bus controller. Now the flexibility which is inherent in the partitioning of distributed processing means that decisions taken at an early design stage for a system will have an effect on the volume and nature of the intercommunicated data. This, together with the fact that the system data traffic is under software control, demands that an integrated top-down approach is taken to the overall system design. Thus, it is important to investigate the total system architecture at an early stage in the project so that the individual subsystem requirements can be hierarchically derived from a common base.

At the outset of a design study, then, it is valuable to postulate, in a reasonable amount of detail, the operational functions to be performed by the system, and the nature of the subsystems which comprise these functions, together with estimates of the data flowing between the functional areas. Given this initial system breakdown, it is then possible to subject it to analysis in order to obtain an early indication of the correctness of the approach. Important factors to check can include:

- Conformance - whether the postulated design functionally conforms to the requirement.
- Consistency - whether data produced or required by one subsystem is consistent with the capabilities or requirements of other subsystems, whether there are conflicts in the production of data, and so on.
- Completeness - whether all required subsystems exist, whether all required data is generated and all generated data is used, etc.
- Feasibility - whether requirements placed on subsystems are within their capabilities, whether total system data flow produces acceptable data bus loading estimates, and so on.

It is clear that the earlier the stage of development at which problems can be identified, the less they cost to resolve. It is also true that the more complex the proposed system, the greater will be the potential benefits of early system design analysis. At the same time, the very complexity which prompts this approach may result in a design analysis procedure which is extremely tedious, time-consuming and error-prone in itself unless automatic methods involving computer assistance are adopted.

The systematic analysis of a large database is, of course, a task ideally suited to a digital computer, which has an infallible memory and inexhaustible patience. Furthermore, once the decision is made to invoke automatic assistance, then further benefits become apparent. As well as using the computer to trace errors and inconsistencies in the proposed system, the existence of the database facilitates the trying out of different configurations, trade-offs, etc, and the examination of the subsequent effects in an iterative manner which would normally be too time consuming if done manually. It can also provide an automatic documentation service on the current and previous system configurations, and the forms of the reports can be many and varied according to the needs of the consumer.

Once the system database has been processed automatically, further manipulation can be capable of tuning the resultant configuration into a form acceptable to the system designer, and the specification of the system data traffic which resides in the database can be used automatically to generate bus control schedules and subsystem interface requirements.

This paper, then, describes such a system analysis program which has been developed at RAE Farnborough specifically to investigate problems of completeness, consistency and feasibility for a postulated avionic system design. Its application is oriented towards loosely-coupled bus-connected systems but it is not exclusively confined to these, and flexibility has been built into the program by including resetable parameters which specify aspects of the intercommunication philosophy.

The technique is called SAVANT, which is an acronym for System Architecture Verification and Analysis Technique. It is programmed in Coral 66 and was developed on a Prime 300 computer system. A design aim was to make the program as transportable as possible, using no machine dependent features in the main body of the source by extensive use of macro definitions. SAVANT is now operational on a PDP 11/34 system in addition to the Prime 300.

2 PREPARING TO USE SAVANT

SAVANT is an interactive program, which means that the user communicates with the program by means of a VDU terminal and keyboard, giving commands to the program which define the required operations and responding to prompts and questions displayed in order to amplify or qualify the commands. Results and reports are received directly on the terminal as well as being able to be committed to file for future reference and hard copy output.

The purpose of SAVANT is to provide an automatic tool to enable the data flows in a speculative system design to be analysed and refined in an iterative manner so that errors and inconsistencies can be corrected, the effects of various trade-offs can be examined and aspects of the feasibility of the proposed design can be established at an early stage.

The SAVANT program operates on a database held in memory. The database represents various facets of the system under investigation and SAVANT provides the facilities for creating, modifying, extending, analysing, filing and retrieving the data. The database is divided into three segments: the reference segment, the configured system segment and the messages segment.

The 'raw' system data which the user prepares forms the basis of the reference segment of the database. In order to generate this, the user formulates a list of potential subsystems which may be included in the system under investigation, although the term 'subsystem' is very much dependent on the interpretation which the user wishes to place on it. For example, it may be a single identifiable subsystem, or it may be a complete functional area which in reality would consist of a number of distinguishable subsystems. On the other hand, several different 'subsystems' might in fact be different manifestations of the same subsystem allowing for different modes of operation. Once the data is on the SAVANT database, a 'system' can be configured from whichever of the known 'subsystems' the user wishes to nominate, and not necessarily all of them.

Each subsystem is given a name which describes its function, such as 'INU' (inertial navigation unit), 'ADC' (air data computer), 'HUD 1' (one option of the head-up display subsystem) and so on. Having decided on the subsystems, the user then prepares a list of all the data items which are required to be received by each and the data items which each will produce for transmission to other subsystems, and each of these data items is given a name. Thus, the INU might require to receive 'BARO ALT' and 'MACH' among its input data, and may produce 'LATITUDE' and 'LONGITUDE' among its output data.

Each data item thus specified must be provided with an estimation of its iteration (update) rate and its precision, either required, if it is input by the subsystem, or capable of being produced if it is output by the subsystem. Also, the units in which the data is represented may be specified.

For the representation of rate, SAVANT uses the 'rate group' concept rather than absolute iteration rates. By this method, the maximum data iteration rate in the system (which might, in practice, be 50, 64 or 100 Hz, say) is represented by Rate 1. Binary subdivisions of this rate are then expressed as Rate N, so that Rate 2 is one half of Rate 1, Rate 3 is one half of Rate 2, and so on. The decision about absolute rate values does not have to be made at this stage, and one of the SAVANT analyses can be to investigate the effect on system data traffic of varying the value of Rate 1. Rate 0 is used to represent a direct connection where the data is not transmitted on the data bus.

The precision value is simply the number of bits needed to represent the data quantity. Thus, accuracy, range and resolution are comprehended within this figure, but it is felt that, together with the units identification, this is adequate to express the precision attribute of the data quantity at this stage of the description without introducing undue complexity.

The units identifier, like the subsystem and data name, is an alphanumeric character string. If a data quantity is dimensionless, such as a ratio, or if units are not considered important to the analysis, then the string can be null.

To summarise, then, in preparation for operating SAVANT, the user has formed a description of the speculative system which comprises a number of data flow specifiers, each of which consists of:

- Subsystem name
- Data item name
- Data flow direction (transmitted or received)
- Data rate
- Data precision
- Data units

and it is such data flow specifiers which form the basis of the reference segment of the database. The way the data is used in analysing the system design is described in the next section.

3 THE OPERATION OF SAVANT

Depending on the phase of operation and the state of the database, the SAVANT program operates in one of seven program states. These program states govern the tasks which the user is able to perform. The seven states in which the user can operate are as follows:

EMPTY
OPEN
INCONSISTENT
CONFIGURED
FORMED
LIMITED
LIMITED FORMED

The full range of commands to which SAVANT responds comprises 52 major commands, some of which can be further qualified in operation. Of this range, only a certain number are valid in each state, and the user can at any time display the current program state and the menu of commands valid in that state by typing 'H'. If the user types a command which is not valid in the current state, or is simply not understood, SAVANT comments on the fact, echoing the erroneous command and reminding the user of the listing option.

With the exceptions of the commands 'H' and 'STOP', all major commands which the user types consist of 3-letter abbreviations of the actual commands. A list of all commands, showing the abbreviations and the states in which they are valid, is given in Table 1.

Often during the course of command execution, the user is requested to supply further information, such as subsystem or data names, file names, and the like. In such cases, if the user's response is erroneous, such as giving a name which does not exist, or specifying a file for reading which contains the wrong type of information, the user is warned of the error and SAVANT returns to the command level. The program always recovers from error in this manner and never exits without giving the user a chance to file the data on which he has been working.

The operation of SAVANT in each state will now be described.

3.1 The EMPTY state.

When the SAVANT program is started, the database area is initialised and the program enters the EMPTY state. An operation which can only be done in this state is to reset any or all of the preset system parameters. These are declared in SAVANT with the values characteristic of Mil Std 1553B, where appropriate, so every time the program is run afresh then these values will prevail, but once they are changed within a run then the new values prevail until changed again or until the program is stopped.

The parameters which can be changed are as follows, with the preset values given in parentheses: lowest rate group (8), data word length (16 bits), number of words in a message (32), word overhead - eg, sync, parity, etc - (4 bits), transmission bit rate (1 Mbit/s), number of addressable terminals (30), typical message overhead for transfers involving the bus controller (2.6 words) and not involving the bus controller (4.9 words).

The other operations which can be performed in the EMPTY state involve the input of data, either directly from the terminal or from a disc file. Data input from the terminal comprises information on the data flow specifiers detailed in Section 2, and this is entered in the reference segment of the database. Multiple entries can be made with one command, and these can either be miscellaneous or specific to one subsystem. In the latter case the subsystem name need only be typed once. The user is prompted for each specific piece of information required. As soon as an operation is performed which places data in the reference segment then the program state becomes OPEN.

It will be seen later that a reference database which exists within SAVANT can be saved as a disc file, and such a file can also be input in the EMPTY state to set up the reference segment. Again, this results in the program state becoming OPEN.

Another option in the EMPTY state is to input information from a disc file directly into the messages segment of the database, in which case the program state becomes LIMITED. This is discussed in 3.6.

3.2 The OPEN state.

The OPEN state allows various operations to be performed on the reference segment. These fall into several categories: listing, extension, modification, filing and state-changing.

The listing commands allow various lists to be produced. For example, one can list the reference database entries in tabulated form, or one can list only those entries which relate to a specific subsystem. One can produce a list of all subsystem names or a list of all data item names. Also, one can trace a data item by listing all occurrences of that name in the reference segment, with the appropriate qualifying information.

With all of the listing commands in SAVANT, if the list produced could exceed the capacity of the terminal screen then the user is offered the option to pause on each page so that the information can be examined at leisure. This option also allows the output then to be aborted rather than continued to the next page. If the option is not exercised then the listing runs to completion without pausing, which may be useful if a monitor file is being produced. All the listing commands of the OPEN state are also available in the INCONSISTENT, CONFIGURED and FORMED states.

The reference segment can be extended by adding entries from terminal or file, using the same commands as are available in the EMPTY state.

Modification of the reference entries can take several forms. Any subsystem or data item name can be changed, either to a completely new name or to another name which already exists on the database. This latter is useful for resolving spelling inconsistencies. Rate and precision entry values can be modified in the following ways: the value for a specific entry can be changed, the entry being identified by the subsystem and data names and the transmit/receive flag; the value for a particular data item can be generally set/changed at every occurrence of the data item, and any specified rate or precision value can be changed to another value either generally throughout the reference segment, or just for those entries relating to a specified subsystem.

A units identifier can also be changed to a new or other existing name either generally throughout the reference segment, or for every occurrence of a data item, or just for one specific entry.

Finally, among the modification commands, reference segment entries can be deleted in two ways. Either a specific entry, identified by its subsystem and data names and transmit/receive flag, can be deleted, or all entries relating to a specified data item can be deleted. If any deleting operation removes the last remaining reference segment entry then the program state reverts to EMPTY.

The filing commands enable the current reference segment to be saved in one of two ways. Either the complete reference segment can be filed or only those entries relating to a specified subsystem. Using the latter command, a library of files relating to different subsystems can be established. When the files are written they are provided with identification which can be checked as part of the reading-back operation.

There are two state-changing commands available in the OPEN state. Firstly, the database can be cleared, in which case the program state reverts to EMPTY. The user is asked to confirm the intention to clear since any work thus far performed will be lost if no filing has taken place. The second state-changing command is that which requests a system to be configured from the subsystems known to the reference segment. Here the user can specify that all subsystems are to be included, or a 'yes/no' indication can be given as SAVANT offers each subsystem in turn.

Once SAVANT has ascertained the subsystems to be included in the configured system it formulates the configured system segment of the database, using information derived from the reference segment. This configured system segment contains information on the desired linking of data in the system, identifying all transmitted data items, together with their rate, precision and units capabilities, and all configured receivers of each data item, together with their rate, precision and units requirements.

The data thus assembled is then checked for fatal inconsistencies - ie, those which preclude the formation of valid messages between subsystems in the configured system. If such exist then the messages segment of the database cannot be formulated and the program state becomes INCONSISTENT. If no fatal inconsistencies exist then the messages segment, which comprises the valid traffic resulting from the data linking operation, is formulated and the program state becomes CONFIGURED.

In neither of these states can the reference segment of the database be modified, since this must always correlate with the system which has been configured.

3.3 The INCONSISTENT state.

If the program state on configuration is found to be INCONSISTENT, then SAVANT automatically generates a list of the fatal inconsistencies which have been found. These fall into four categories. Firstly, the rate, precision or units requirement specified for a data item in a particular receiver may not be consistent with the capability specified for the transmitter of the data item (ie, rate or precision too high, or different units). Secondly, it may be found that a data item is transmitted by more than one subsystem, or, thirdly, that a subsystem both transmits and receives the same data item. The fourth type of fatal inconsistency is where the number of addressable terminals resulting from the data traffic would exceed the terminal limit set.

Any of these would preclude the formation of valid message traffic in the system, and the list is useful in identifying where the reference segment needs to be corrected. Of course, such modification can only be performed in the OPEN state, so the only state-changing command available in this state is to dismantle the inconsistent configured system, in which case the state reverts to OPEN. No modification to any of the database structures can be performed in the INCONSISTENT state. The listing and filing options of the OPEN state are still available, and there are now three further listing commands.

One of these commands allows the user to re-display the list of fatal inconsistencies, and another is used to generate a list of non-fatal inconsistencies, indicating a lack of completeness of data paths. Here all data generated and not used in the configured system is listed, as is all data required but not generated. This type of incompleteness is not fatal because it does not provide any dilemma in forming the messages - the incomplete path is simply not included in the message structure - and it may be an intentional condition of this configuration. The third new list command produces a general trace of data in the configured system, listing each transmitted data item, together with its transmitting subsystem and capabilities and all configured receivers with their requirements. The latter two listing commands are also valid in the CONFIGURED state - the former is not applicable since there are no fatal inconsistencies if the program is in that state.

It should be noted that the STOP command is valid in all program states. This stops the SAVANT run and returns control to the computer operating system. Before the command is executed the user is requested to confirm the intention to stop since all information on the database will be lost unless filing has taken place.

3.4 The CONFIGURED state.

When a system is configured from the reference segment information which does not involve any fatal inconsistency then the program state becomes CONFIGURED. As well as formulating the configured system segment, SAVANT is also able to set up the messages segment of the database.

A message is a package of data words passing from a specific source subsystem to a specific sink subsystem (a source/sink pair) at a specific rate. During configuration by SAVANT, if the total number of data words satisfying this requirement in a particular case exceeds the value of the message length setting then more than one actual message must be generated. Furthermore, if the precision requirement for a data item is greater than the word length setting then more than one data word must be used to represent the quantity. This is known as partitioning of data words. In SAVANT, the precision value of a data item can be as high as 1024 bits (64 x 16-bit words), so it is possible, when formulating the design, to comprise a large package of data under one generic name.

The partitioning of data words and the disposition of words into messages is performed automatically during configuration, and the result forms the basis of the messages segment. There is one entry for each message generated, and each entry contains references to the source and sink subsystems, rate, number of words and the individual data word names. Partitioned data words are assigned subscripts so that they can be individually identified during modification. Such modification can be performed, and further information added to the messages segment, by commands available in the CONFIGURED state.

The operations which can be performed in this state fall into several categories: listing, analysis, modification, filing and state-changing. As well as the listing commands available in the OPEN and INCONSISTENT states, a new range of listing options become available in the CONFIGURED state. For example, it is possible to list the names of configured subsystems, and this list includes indications as to whether each subsystem is connected via the data bus and, if so, whether its terminal address has been set and what the value is. The terminal address is required by Mil Std 1553B protocols, and one of the commands in this state provides for the assignment of these values by the user.

Several listing options are derived from the configured system segment of the database. For example, one can produce a list of all source/sink pairs together with all data items passing between each, each data item being qualified by rate value, or one can list this information for one specified source/sink pair. One can also generate a visual map of the data traffic between source/sink pairs, either displayed on the terminal screen or directly into a file.

Other listing commands derive their information from the messages segment. A summary list of messages can be generated which tabulates, for each message, the source, sink, rate, number of words and retry code. The retry code is intended to be used for error recovery action by the bus controller in the actual system as implemented and, since SAVANT may be used to generate bus control schedules, the user is able to assign or change values of this code for any message whilst in the CONFIGURED state. In addition to the summary, fuller details of all bus messages and all direct data packages can be listed, including identification of the actual data words, or, alternatively, one can list this information for any one specific message which needs to be examined.

The analysis commands enable the user to analyse the message structure for the presence of message subsets, and to calculate the data bus loading percentages which would result from implementation of the system in practice.

The subset information is required so that the user may rationalise the message structure in preparation for the automatic generation and assignment of subaddresses which takes place when the program is advanced to the FORMED state. A subaddress is used to identify a particular message, or package of data within a subsystem. In other words, the subaddress value can be used as a vector which accesses the beginning of the package of data within the subsystem. This is particularly valid if a subsystem involves processing and is likely to buffer its data in memory. In the Mil Std 1553B protocol the subaddress value is part of the command word which is sent by the bus controller to a subsystem, and it is likely that the subaddress for a particular message will be different in the source and sink subsystems.

Naturally, if a subsystem transmits an identical message -- in terms of data content -- to a number of different receivers, then the contents of that message can be assigned a single subaddress for the source subsystem. Furthermore, if the message sent to Terminal 1, say, is a subset of the message sent to Terminal 2 then the subaddress for the shorter message within the source subsystem can still be the same as that for the longer message since the word count field of the command word specifies the number of words in the message. For this to be valid, bearing in mind that the subaddress points to the beginning of the data, the words comprising the short message must be contiguous within the longer message and must occupy the beginning of that message.

The facility to list subsets is, therefore, provided to allow the user to examine the relevant messages and make appropriate use of the modification commands in order that the automatically assigned subaddresses are as efficiently derived as possible.

An important measure of the feasibility of a proposed system is whether the data bus has the capacity to handle all the required data traffic. The loading calculation command causes SAVANT to estimate the loading percentages which would result from running the system in reality, using the set values for overheads and transmission rate. The user is requested to specify the value in Hz represented by Rate 1 and to declare which subsystem is to act as bus controller. If none of the known configured subsystems is indicated then SAVANT assumes a dedicated controller which is not taking part in the actual message traffic. SAVANT calculates and displays the load in words, including overheads, at each rate, and then displays three bus loading percentages.

In order to comprehend these it is necessary to define the 'major frame' as the interval represented by the lowest iteration rate in the system -- ie, the iteration period of the complete message repertory,

and the 'minor cycle' as the highest iteration rate period. The first loading figure calculated, then, is the long-term (major frame) average load, and then two peak (minor cycle) loading figures are displayed. The first of these is calculated on the basis that all rate group messages are initiated in the same minor cycle - eg, when the system is reset on start-up, say - and this is the peak lumped loading. The peak distributed loading, on the other hand, assumes that the initiation of the different rate group messages is staggered so that only the Rate 1 messages and those in one of the other rate groups occur in any minor cycle. Thus, it may be possible to observe, for example, that a system whose average loading is acceptably low would produce an impossibly high peak lumped loading percentage which is alleviated by distributing the initiation of the different rate group messages.

The use of the loading analysis on the configured system, particularly as the message structure is modified, or as the reference segment is rationalised and the system re-configured, provides a vital check on the feasibility of the design approach.

Two of the modification commands have already been mentioned - the ability to set and change terminal addresses and retry codes. Two further modification commands are available in order to allow the user to change the order of data words within messages or to redistribute data words between messages. SAVANT formulates message contents in the order in which data items are encountered in the reference segment, the result of which may not be satisfactory to the user. For example, it may be necessary to re-order a message in order to rationalise subsets, or partitioned data may cross the boundary between two messages whereas the designer would prefer it in one message, and so on.

There are three filing commands available in the CONFIGURED state. The two reference segment filing options of the OPEN state are still valid, and it is now also possible to file a message structure for future reference. In this case, only the information in the messages segment of the database is sent to file, but this includes any terminal address and retry code assignments which might have been made and, of course, takes account of any message re-formatting.

Of the two state-changing commands, one dismantles the configured system, taking the program back to the OPEN state, thus facilitating further reference segment modification. The other advances the program to the FORMED state.

3.5 The FORMED state

In the FORMED state the database represents a finalised system configuration and is, therefore, no longer available for modification. It is from this state that the user is able to generate schedule tables for control of the system in reality, together with detailed information on data requirements in a subsystem related form.

When the command is given to form the system, SAVANT automatically generates the source and six subaddresses for each message, making the best possible use of subset information, and implants these values in the messages segment entries. There is a limit to the maximum number of subaddresses for a subsystem: in Mil Std 1553B, for example, this limit is 31. Thus if, during the course of formation, a message subaddress value for any subsystem would exceed this limit then the user is warned, the forming fails and the program remains in the CONFIGURED state.

This, then, is another check on the feasibility of the proposed system, and if forming fails then it is up to the user to examine the message structure and the reference segment data in order to reduce the total number of messages a subsystem has to handle. One way, for example, may be to reduce the number of different rate group transfers so that messages may be coalesced.

The commands available in the FORMED state include all the listing and filing commands of the OPEN and CONFIGURED states but there are no database modification commands of any sort. There is one new listing command which displays a list of all subaddress assignments, and one new filing command which generates the bus control schedule.

On receipt of this latter command, SAVANT requests the user to declare which subsystem is to act as the bus controller and, as in the calculation of bus loading, assumes a dedicated controller if the name supplied is not recognised. From the information in the messages segment, a table of bus control schedules is created and sent to a disc file. The format for the schedule is flexible and can be changed to suit any particular bus controller implementation although, at the present time, the flexibility is not parameterised. It is intended that this should be so in the future.

Following the schedule filing, SAVANT then sends to another disc file details on each subsystem's requirements, including its terminal address, the data content of all subaddresses, transmitted and received, and all direct transfer information.

The only state-changing command in the FORMED state reverts the program to the CONFIGURED state.

3.6 The LIMITED and LIMITED FORMED states.

It was discussed in 3.4 how the message structure of a configured system can be saved as a disc file. With SAVANT in the OPEN state, such a file can be read back into the messages segment, in which case the program state becomes LIMITED, which is a special case of the CONFIGURED state.

This is because although the messages segment exists it is backed up by neither the reference segment nor the configured system segment, so the range of commands available is constrained. For example, the only listing commands are those related to the messages segment: the listing of configured subsystems, message summary, message details and the data traffic map, plus the listing of data item names. It is not possible to revert to the OPEN state because there is no reference segment. The only reversion from the LIMITED state is directly to the OPEN state.

In the LIMITED state, however, the analysis and modification commands of the CONFIGURED state are still available, so it is possible to modify the message structure, to perform loading analyses, to assign terminal addresses, etc. It is also possible to form the system, in which case the program state becomes LIMITED FORMED. Thus bus control schedules for the revised message structure can be generated. Reversion from the LIMITED FORMED state is to LIMITED.

Figure 1 shows the relationship of all the SAVANT program states.

4 CONCLUSIONS

This paper has briefly described the software technique for system analysis known as SAVANT. A full report on its development and application, including an example of its use, has been published (ref 6).

Future avionic systems are certain to become more integrated, where the individual subsystem elements must be regarded as performing co-operatively in order to provide their overall contribution to the system task. The options open to the initial system designer are varied, and decisions made at an early stage have a very significant effect on the future development of the system.

The volume of information which has to be considered at this early design stage, and the nature of the analysis tasks which have to be done, demand that tools and techniques are developed which enable automatic processing to play the part for which it is so clearly suited. The availability of such tools and the contribution they can provide in easing the more menial design tasks in an unerring manner mean that the initial design phase can be more speculative, trying out different ideas and gauging their effects. This can only be of benefit to the resulting design.

The SAVANT technique described in this paper provides a facility which, it is believed, will prove a useful part of the standard warehouse of support tools needed for the development of future avionic systems.

Copyright ©, Controller HMSO, London 1981

REFERENCES

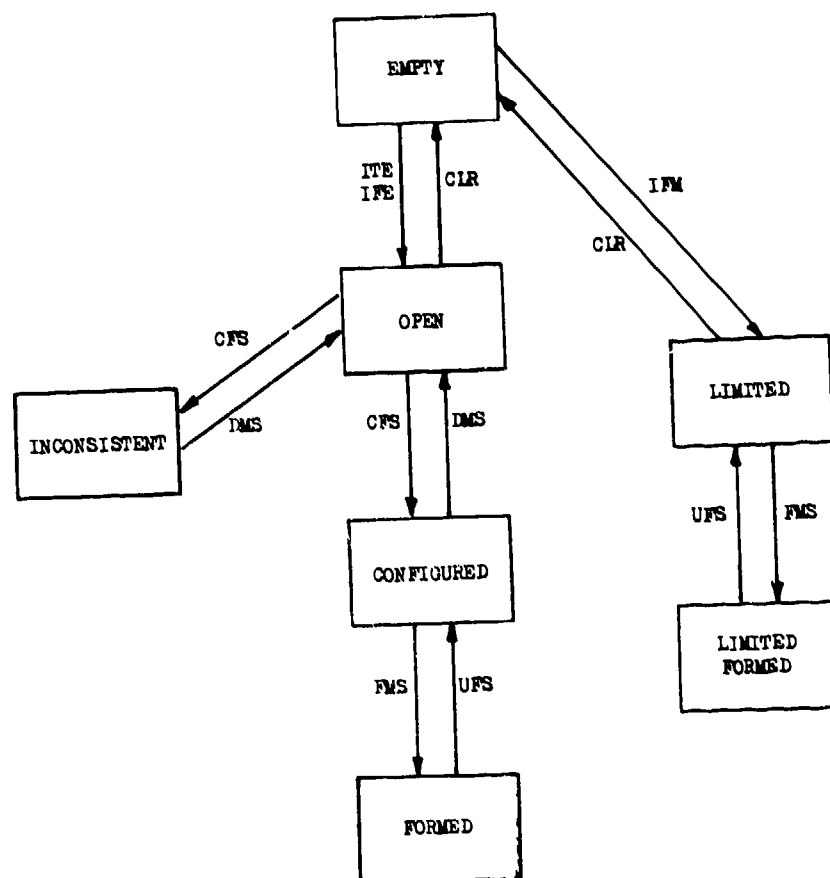
- 1 E C Gangl Time-division multiplexed data bus integration techniques. Proceedings, USAF Multiplex Data Bus Conference. Dayton, Ohio, November 1976.
- 2 A A Callaway Trends in digital data processing and system architecture. AGARD OCT Conference Proceedings 272. Ottawa, May 1979.
- 3 A A Callaway The influence of digital techniques on system architecture. Joint IEE/RAeS Symposium: Digital Avionics, promise and practice. London, March 1980.
- 4 US DOD Aircraft Internal Time Division Command/Response Multiplex Data Bus. Mil Std 1553B. 21 September 1978.
- 5 Ministry of Defence Serial time division command/response multiplex data bus. Def Stan OO/18 (Part 2) Issue 1. 28 April 1980.
- 6 A A Callaway SAVANT - A database manipulation technique for system architecture verification and design analysis. RAE TR80101. January 1981.

Table 1
SAVANT COMMANDS

<u>Command</u>	<u>Valid states</u>
H - LIST VALID COMMANDS	E . O . I . C . F . L . LF
RSP - RESET PARAMETERS	E
ITE - INPUT TERMINAL ENTRIES	E . O
IFE - INPUT FILED ENTRIES	E . O
IFM - INPUT FILED MESSAGES	E
LEN - LIST ENTRIES	O . I . C . F
LSB - LIST SUBSYSTEM DATA	O . I . C . F
LSN - LIST SUBSYSTEM NAMES	O . I . C . F . L . LF
LDN - LIST DATA NAMES	O . I . C . F . L . LF
TDP - TRACE DATA PATH	O . I . C . F
CSN - CHANGE SUBSYSTEM NAME	O
CDN - CHANGE DATA NAME	O
CRE - CHANGE RATE ENTRY	O
CRQ - CHANGE RATE GENERAL	O
CDR - CHANGE DATA RATE	O
CSR - CHANGE SUBSYSTEM RATE	O
CPE - CHANGE PREC ENTRY	O
CPO - CHANGE PREC GENERAL	O
CDP - CHANGE DATA PREC	O
CSP - CHANGE SUBSYSTEM PREC	O
CUE - CHANGE UNITS ENTRY	O
CUG - CHANGE UNITS GENERAL	O
CDU - CHANGE DATA UNITS	O
DDR - DELETE DATA REFERENCE	O
DOE - DELETE ONE ENTRY	O
FEN - FILE ENTRIES	O . I . C . F
FSD - FILE SUBSYSTEM DATA	O . I . C . F
CLR - CLEAR DATABASE	O L
CPS - CONFIGURE SYSTEM	O
LIP - LIST DATA PATHS	I . C . F
LCS - LIST CONFIGURED SUBSYSTEMS	I . C . F . L . LF
LDT - LIST DATA TRAFFIC	C . F
MDT - MAP DATA TRAFFIC	C . F . L . LF
SSP - SOURCE SINK PAIR DATA	C . F
CCS - CHECK CONSISTENCY	I
CCR - CHECK CORRELATION	I . C . F
SUM - SUMMARISE MESSAGES	C . F . L . LF
LEM - LIST BUS MESSAGES	C . F . L . LF
LDD - LIST DIRECT DATA	C . F . L . LF
CLD - CALCULATE LOADINGS	C . F . L . LF
LOM - LIST ONE MESSAGE	C . F . L . LF
LSS - LIST SUBSETS	C . F . L . LF
SRC - SET RETRY CODE	C . . . L
STA - SET TERMINAL ADDRESS	C . . . L
ROM - REORDER MESSAGE	C . . . L
RPM - REFORMAT MESSAGES	C . . . L
FLM - FILE MESSAGES	C . F . L . LF
DMS - DISMANTLE SYSTEM	I . C
FMS - FORM SYSTEM	C . . . L
LSA - LIST SUBADDRESSES	F . . . LF
FLS - FILE SCHEDULES	F . . . LF
UPS - UNIFORM SYSTEM	F . . . LF
STOP (return to OS)	E . O . I . C . F . L . LF

Key : E - EMPTY
O - OPEN
I - INCONSISTENT
C - CONFIGURED
F - FORMED
L - LIMITED
LF - LIMITED FORMED

Figure 1
SAVANT Program State Diagram



Key to commands:

ITE - INPUT TERMINAL ENTRIES
 IFE - INPUT FILED ENTRIES
 IFM - INPUT FILED MESSAGES
 CLR - CLEAR DATABASE
 CFS - CONFIGURE SYSTEM
 DMS - DISMANTLE SYSTEM
 FMS - FORM SYSTEM
 UFS - UNIFORM SYSTEM

SIGNAL PROCESSING WITH SYSTOLIC ARRAYS*

R.W. Priester
Research Triangle Institute
Research Triangle Park, NC 27709

K. Bromley
Naval Ocean Systems Center
Catalina Boulevard
San Diego, CA 92152

H.J. Whitehouse
Naval Ocean Systems Center
Catalina Boulevard
San Diego, CA 92152

J.B. Clary
Research Triangle Institute
Research Triangle Park, NC 27709

ABSTRACT

This paper discusses the application of systolic array processors to signal processing problems that are amenable to a matrix formulation. Systolic arrays are formed by providing nearest-neighbor interconnections between a large number of elemental processors to form either a one- or two-dimensional array. With the possible exception of boundary elements, each processing element performs identical computations in synchronism with other elements in the array. A number of important problems for which systolic arrays hold potential are mentioned and the systolic array processor definition, in a number of its forms, is reviewed. When applied to strongly band-limited matrices, systolic array processors can be characterized as highly efficient from the standpoint of both hardware utilization and algorithm time. However, as the bandwidth becomes large, this high performance is degraded. In an effort to overcome performance degradation, this paper introduces and evaluates a data transformation which, when applied to an $n \times n$ dense matrix, results in an improved banded structure with attendant hardware savings. An interesting feature of this transform is its invariance properties with respect to the ordering of output time sequences and algorithm execution time. Another interesting aspect is its relation to the classical Gauss-Seidel's method of iteration.

It is shown that systolic array processors possess some efficient testability features which can be exploited concurrently. These are briefly summarized.

*The work reported in this paper was sponsored by the Naval Ocean Systems Center, San Diego, CA, under contract N66001-80-C-0118.

1.0 INTRODUCTION

This paper discusses the application of systolic array architectures to signal processing problems.

Introduced by Kung (1978), systolic array architectures provide the capability for realizing a number of important matrix operations. In addition to achieving a high computation rate by means of pipelining and concurrent computation, the architecture is a good candidate for implementation with VLSI (very large scale integration) technology. If the matrices processed are characterized by a narrow bandwidth, excellent hardware utilization efficiency can be achieved. However, in those cases where the matrix bandwidth becomes appreciable, for instance in the case of square densely-populated matrices, hardware utilization efficiency is degraded significantly. This paper addresses the problem of using systolic arrays to process matrices whose structure is less constrained. A simple but effective data transform which can in some instances significantly improve hardware utilization efficiency is introduced and developed.

The paper is organized as follows. Section 2.0 presents a brief and general discussion of several problem areas where the systolic array architecture is of interest. Section 3.0 outlines the main features of the systolic array architecture and only summarizes the extensive treatment given by Kung (1978) and Mead (1980); this section is included only for purposes of completeness of presentation. The PRT (partial row translation) data transform is introduced and developed in detail in Section 4.0. Section 4.0 also quantitatively compares the efficiency of the original systolic array processor with that which results from applying the PRT transform. These results provide a means for deciding when PRT is advantageous. Matrix inversion is the topic of Section 5.0 while Section 6.0 briefly outlines an efficient technique that is useful for testing some systolic array matrix processors.

2.0 MATRIX OPERATIONS IN SIGNAL PROCESSING APPLICATIONS

Matrix operations represent a significant portion of the computational burden encountered in many signal processing applications. Adaptive filtering, data compression, beamforming, and cross-ambiguity calculation represent problem areas where stable matrix analysis techniques are of current interest. In terms of resources required for system implementation, these problems can be classified as memory intensive and computation intensive. Construction of systems capable of providing the computations required for analysis of the above problems must provide for operations such as matrix multiplication, inversion, addition and various decompositions.

For example, in least squares approximation problems, one might encounter matrix multiplication, matrix inversion, and/or singular value decomposition. The computational approach used in a particular instance depends upon the numerical stability properties of the problem at hand. For instance, if the order of a particular problem is sufficiently small, the Gauss normal equations might be solved by performing a straightforward matrix inversion. However, in the solution of ill-conditioned systems commonly encountered in large-scale problems, achieving a meaningful solution might require application of singular value decomposition computations.

Speiser and Whitehouse (1980) discussed the signal processing problems mentioned above and considered the applicability of competing architectures such as transversal filters, array processors, bus-organized multiprocessors and systolic array architectures. Of these, the most promising architecture is that of the systolic array which has the potential to support real-time implementation of the algorithms required in order to address those problem areas mentioned in this section.

3.0 THE SYSTOLIC ARRAY ARCHITECTURE

In the interest of a self-contained presentation, the systolic array architecture will be outlined and illustrated in this section. A thorough, comprehensive treatment can be found in Kung (1978) or in Mead (1980). The systolic array architecture is founded almost exclusively upon a single computational element - the inner product step processor - which implements the relation

$$y^{k+1} = a_{k+1} \cdot x_{k+1} + y^k; \quad k = 0, 1, 2, \dots, n-1. \quad (1)$$

Systolic array processors are constructed by appropriately interconnecting a group of inner product step processors. In the systolic array architecture, only nearest-neighbor processor communication is permitted. For purposes of data communication and computation, each inner product step processor is equipped with three data registers: R_y (for y), R_a (for a_k) and R_x (for x_k). Each register has two connections - one for input, the other for output. Kung (1978) defined two types of inner product step processors which are illustrated in Fig. 1. These elemental processors can be connected in a number of ways which provide the capability to perform various matrix operations such as matrix multiplication, LU decomposition of symmetric positive-definite matrices, and the solution of triangular linear systems or equations.

A basic unit of time measure for both types of processors shown in Fig. 1 is defined as follows: (a) the processor loads inputs y^k , x_k and a_k into R_y , R_x , and R_a respectively, (b) y^{k+1} is computed according to (1), and (c) y^{k+1} , x_k , and a_k are output.

As an example, a systolic array matrix-vector processor will be configured to form the product

$$y = Ax \quad (2)$$

using a linearly connected group of Type 1 processors. The relations which must be implemented are as follows

$$y_i^{k+1} = a_{i, k+1} \cdot x_{k+1} + y_i^k, \quad k = 0, 1, 2, \dots, n-1$$

$$y_i^0 = 0 \quad (3)$$

$$y_i = y_i^w, \quad i = 1, 2, \dots, n.$$

Fig. 2 illustrates the systolic array of processors, the element data arrangements and flow required to evaluate (2) for the case where A is an $n \times n$ matrix with bandwidth $w = p + q - 1 = 4$. The y_i enter the array from the right as zero and accumulate so as to form the inner product of the i th row of A with vector x which moves to the right after being input from the left. As the x and y vectors move through the array in the manner noted, A is shifted downward such that elements along the main diagonal pass through P_2 . In general elements of A above and parallel to the main diagonal pass through processors to the left of P_2 . Similarly elements of A below and parallel to the main diagonal pass through processors to the right of P_2 . A detailed example illustrating the operation of this systolic array matrix-vector processor will be presented in Section 4.0.

Generalization of the linearly-connected systolic array to a two-dimensional orthogonally-connected structure enables the evaluation of matrix-matrix products. A systolic array for evaluating

$$C = AB \quad (4)$$

where all matrices are $n \times n$ is shown in Fig. 3. Matrix A is input to the systolic array in exactly the same way as described earlier for the matrix-vector processor while columns of B are input, with appropriate spatial shift to allow for A's time delay, into successive rows of the array. If B contains a large number of columns this implementation can be inefficient even for strongly banded matrices. Kung (1978) overcame this problem by devising the hexagonal-connected systolic array which is based upon the Type 2 processor of Fig. 1. An example of this processor is presented in Fig. 3 (b) for the case (4) when A, B and C are strongly banded. Note the direction of flow and orientation of A, B and C. Entries in C are accumulated as this matrix is shifted upward from the bottom of the array, where the c_{ij} enter with zero value.

Using the array structures presented above, Kung (1978) was able to realize two additional important matrix operations. Due to space limitations, these only will be mentioned here. A triangle equation solver can be constructed using a linearly connected array of inner product step processors; however, it is necessary to introduce a new processor capable of division. The resulting processor solves a nonsingular triangular system of linear equations by back-substitution. Similarly, by adding special elements on the upper portion of the periphery of the hexagonal array (Fig. 3b), Kung (1978) showed that one can obtain the following matrix decomposition

$$A = LU$$

where A is a symmetric, positive definite matrix
 L is lower triangular having 1s on the main diagonal
 and U is upper triangular.

Therefore, this processor, when coupled with the triangle equation solver, can be used to solve a fairly general class of simultaneous equations.

Table 1 summarizes the hardware requirements and algorithm execution time steps for the family of systolic array processors defined by Kung. When considered from the standpoint of hardware uniformity, a surprising degree of capability is realized by the systolic array architecture. For the case of strongly banded matrix structures, this architecture is efficient in terms of both the quantity of hardware used and in hardware utilization efficiency. However, if square dense matrices or matrices of more general structure are considered, hardware utilization efficiency can be degraded considerably. This problem is

Table 1. Summary of Systolic Array Hardware and Algorithm Execution Time Requirements for Some Matrix Problems.

Systolic Array Configuration	Problem Solved	No of Processors Required	Algorithm Time	Note: (a) Matrices are assumed $n \times n$ with bandwidths $w = p + q - 1$. Subscripted w denotes bandwidth of indicated matrix.
Linearly Connected Array	Matrix-Vector Multiplication	w	$2n + w$	
Linearly Connected Array	Solution of Triangular System	w	$2n + w$	
Orthogonally Connected Array	Matrix-Matrix Multiplication	$n \cdot \min(w_A, w_B)$	$3n + \min(w_A, w_B)$	(b) Matrix-Matrix Multiplication either $C = AB$ or $C' = A'B'$, where $(')$ = transposition.
Hexagonally Connected Array	Matrix-Matrix Multiplication	$w_A w_B$	$3n + \min(w_A, w_B)$	
Modified Hexagonally Connected Array	L-U Decomposition $A = LU$	$p(q-1)$	$3n + \min(p, q)$	

addressed in the next two sections of this paper where methods for improving implementation efficiency are introduced and studied.

4.0 DEFINITION AND DEVELOPMENT OF THE PRT TRANSFORM

In this section the PRT (partial row translation) transform will be defined and some of the benefits available from its application in connection with systolic arrays will be presented. It will be shown to improve hardware utilization efficiency and in addition provide a hardware savings in the case of square dense matrices.

Definition of the PRT Transform

Consider the matrix-vector multiplication problem stated in (2) with A constrained to be $n \times n$ and densely populated. Express A as a strictly subdiagonal part, A_L (i.e. with no diagonal elements) juxtaposed with A_U , the upper triangular part of A which contains the main diagonal elements of A . This may be expressed as follows

$$A = \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix} \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix} \quad (5)$$

Applying the PRT transform to (5) provides

$$A_{PRT} = \begin{bmatrix} & & & 0 \\ & & & \\ & & & \\ & & & \end{bmatrix} \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix} \quad (6)$$

That is, A_{PRT} is obtained from A simply by translating $(i-1)$ elements in row i to the right n positions within the row for $i = 2, 3, \dots, n$. In the resulting $n \times (2n-1)$ array, all elements not specified by A_U and the displaced A_L are set to zero. Now, applying the PRT transform to (2) yields the equivalent expression

$$y = A_{PRT} x_{PRT} = A_{PRT} \begin{bmatrix} x \\ x_p \end{bmatrix} \quad (7)$$

where $x_p = (x_1, x_2, \dots, x_{n-1})$. It is noted that the PRT converts a square array into a non-square array with enhanced banded structure. The transform necessitates augmenting x with a partial copy, x_p . A detailed example where A is 4×4 is shown in Fig. 4. Four processors are used and the required number of time steps is eleven. These quantities compare favorably with Kung's systolic array which would use seven processors and also eleven time steps. For n large, it follows that the PRT transform saves about $n/2$ inner product step processors with no increase in execution time. If the original systolic array were designed such that immediately upon processing element a_{nn} , the values of y contained in the array could be unloaded, a time advantage would result for this processor configuration. The corresponding PRT based array, while saving about one-half the number of processors, would incur only about a 50% increase in execution time.

The PRT transform readily extends to the problem of evaluating the product of two square matrices as expressed in (4). It can be shown that the resulting systolic array for this problem is identical to that of Fig. 3a. The only difference occurs in the way A and B are input to the array. The PRT is applied to A which saves about $n^2/2$ processors and the columns of B , input on the left side of the array are partially repeated as prescribed in (7). Due to the large number of connections which would be required to immediately unload this two dimensional array, the PRT configured processor will evaluate the matrix-matrix product without any time penalty compared with the original systolic array.

Although they will not be discussed here, the PRT transform can be advantageously applied to some problems where non-square matrices are encountered.

Quantitative Assessment of the PRT Transform

The remainder of this section will be devoted to a quantitative comparison of the performance of the systolic array processor proposed by Kung (1978) (hereafter called original and denoted in certain instances by the subscript orig) with that of the PRT based structure (henceforth called alternate and denoted by subscript alt). The comparisons to be made will be based upon the following three figures of merit:

- Processor utilization efficiency η_{orig} and η_{alt} .
- Space-Time product $(ST)_{orig}$ and $(ST)_{alt}$ where
 S = number of inner product step processors
 T = number of algorithm time steps.
- Overall figure of merit $F = \eta/(ST)$, $Q = F_{alt}/F_{orig}$.

In the comparisons which follow, no penalty or cost is assigned to implementing the PRT transform. Also it is assumed that n is large.

First consider the matrix-vector problem which is shown for both processor configurations in Fig. 5. Adjacent to each processor configuration expressions for η , S , and T are given. η is defined as the ratio of active area to the total area as shown in the figure. Simply stated it is an approximate measure of the

proportion of algorithm time for which computations are performed. Only square matrices are considered here with bandwidth $w = p + q - 1$. Note also that the comparisons made here assume processor initialization as illustrated.

Fig. 6 presents plots of η as a function of the normalized bandwidth parameters $y = p/n$ and $x = q/n$. This figure is drawn under the assumption that the array of the original configuration may be unloaded immediately after element a_{nn} has been processed. Alternately, Fig. 7 presents the same information except that immediate unloading of the original configuration is not allowed. The results show that the capability to immediately unload the array is important when $x, y \rightarrow 1.0$. Note that the original configuration provides excellent efficiency for x and y both small, that is, for strongly banded matrices; however, as $x, y \rightarrow 1.0$ the alternate form is superior.

Now consider a comparison on the basis of (ST) product. Solving the relation $(ST)_{org} = (ST)_{alt}$ provides the result plotted in Fig. 8. When the pair (x, y) lie above the curve, the alternate configuration provides a smaller (ST) product.

Generally it will be desirable to maximize the quantity $F = \eta/(ST)$ for a given problem. Therefore, Fig. 9 shows a plot of $Q = F_{alt}/F_{org}$ versus y with x a parameter. Given x and y for a particular problem these results clearly indicate the preferred processor configuration.

Attention is now directed to the matrix multiplication problem where it is required to evaluate $C = AB$ when both A and B are $n \times n$ dense matrices. For the sake of simplicity, the general case of banded matrices will not be treated in this comparison. Three systolic array configurations will be considered.

- (a) A PRT-based orthogonally-connected processor
- (b) The orthogonally-connected processor shown in Fig. 3(a).
- (c) The hex-connected processor presented in Fig. 3(b).

The quantities of interest for comparing these three configurations (subsequently referred to as configuration (a), (b) and (c)) are tabulated in Table 2. (Note in Table 2 that the double subscript on Q is interpreted to mean $Q_{ab} = F_a/F_b$ where a and b refer to the configurations listed above). From these results the PRT-based systolic array is seen to offer significant performance advantages with respect to configurations (b) and (c) under the conditions specified.

Table 2. Comparison of Systolic Array Configurations for Matrix-Matrix Multiplication (all matrices $n \times n$).

Quantity of Interest	Configuration (a)	Configuration (b)	Configuration (c)
T	$5n$	$5n$	$4n$
S	n^2	$2n^2$	$4n^2$
	$2/3$	$1/2$	$1/8$
$Q_{ab} \approx$	2.7		
$Q_{ac} \approx$	17		

5.0 APPLICATIONS OF SYSTOLIC ARRAYS TO MATRIX INVERSION

This section will consider both explicit and implicit methods for solving a given consistent set of linear equations. By explicit it is meant that the inverse matrix is made available to the user while implicit is used to imply that only the solution vector is determined and made available.

The hexagonally connected systolic array mentioned earlier can be used to explicitly invert a given symmetric, positive-definite matrix. The approach is discussed by Speiser and Whitehouse (1980) and can be summarized as follows. First the L-U decomposition of the given matrix is formed using the hex-connected systolic array. Then using n appropriately interconnected triangle equation solvers, L^{-1} can be computed. In this step the input to the array of triangle equation solvers, i.e. the known input vectors taken collectively, forms the identity matrix. U^{-1} is computed in a similar manner, and finally the inverse matrix is obtained by taking the matrix product $U^{-1}L^{-1}$. All of these steps can be implemented using systolic arrays.

Implicit matrix inversion can be performed in several ways, the most direct consisting of L-U decomposition followed by two executions using a triangle equation solver. That is, given

$Ax = b$, A and b known
 $LUx = b$: LU decomposition step
 $Ly = b$: solve for y using triangle equation solver
 $Ux = y$: solve for x using triangle equation solver

This method, while it does not explicitly provide A^{-1} is generally more accurate than the explicit

method which computes $x = A^{-1}b = J^{-1}L^{-1}b$, Sameh (1977). Other implicit techniques such as Jacobi's method, Gauss-Seidel's method and the successive overrelaxation (SOR) method, as discussed by Dahlquist (1974), can be realized with systolic arrays. Implementation of Gauss-Seidel's method is interesting because it is closely related to the PRT transform. Consider the equation $Ax = b$. Factoring A into the form $A = D(L + I + U)$ where L and U are strictly lower and upper triangular matrices respectively (i.e., their main diagonal elements are zero) and D is a diagonal matrix $D = \text{diag}(a_{ii})$, $a_{ii} \neq 0$, $i = 1, 2, \dots, n$.

Jacobi's method of iteration can be written in terms of these definitions as follows

$$x_i^{k+1} = (-L_i x^k - U_i x^k) + b_i/a_{ii}, \quad i = 1, 2, \dots, n \quad (8)$$

where L_i and U_i denote the i th rows of L and U respectively. Implementation of (8) using either the original or alternate forms for systolic array matrix-vector multiplication is straightforward, only requiring insertion of zeros along the main diagonal and evaluation of the terms b_i/a_{ii} outside the array as an auxiliary computation. The equations defining Gauss-Seidel's method are as follows

$$x_i^{k+1} = (-L_i x^{k+1} - U_i x^k) + b_i/a_{ii}, \quad i = 1, 2, \dots, n. \quad (9)$$

Here the notation is identical to that in (8) except that in the term $L_i x^{k+1}$, x^{k+1} represents only a partially filled vector $(x_1, x_2, \dots, x_{i-1}, 0, \dots)$ which is "built up" as the computation proceeds. Gauss-Seidel's iteration can be implemented in systolic array form by using the PRT transform. This is illustrated in Fig. 10 which shows that the diagonal elements have been omitted and the terms b_i/a_{ii} are evaluated outside the array. Assuming that the computation is started with an initial estimate x^k , it can be observed from Fig. 10 that x_1^{k+1} will be output and available for processing by the strictly subdiagonal elements L . (For a detailed example of this property see Fig. 4 and note that in the present case $x_1^{k+1} = y_1$, is output at time step 5. Note also that this value of y_1 is required in time step 6 for processing by a_{21} , which in the present case is L_2). Since U always processes a backdated estimate, it can be seen that the PRT transform, or some equivalent method, must be applied in order to realize Gauss-Seidel's method using systolic arrays. That is, unless the elements of L can be moved to the input side of the array where the x_i^{k+1} are input, the pipelining effect of the array prohibits implementing Gauss-Seidel's method. Therefore, the original form of the systolic array cannot, without modification, be used to implement Gauss-Seidel's iterative method.

Note from Fig. 10 that Gauss-Seidel's implementation can provide extremely efficient utilization of processor capability. Processor utilization efficiency, starting at 83%, monotonically increases toward 100% as the number of iterations increase. Although not discussed earlier when matrix-vector processors were considered, a form similar to that shown in Fig. 10 can be obtained for the problem $y = Ax$ where A is $n \times m$ with $n \geq m$. For this case, input vector x is simply repeated the required number of times while the PRT transform is applied to successive $m \times m$ partitions of A .

The SOR method of solution by iteration is very similar to Gauss-Seidel's method, the most important distinction being that the systolic array in this case computes the residual error which is then weighed by a relaxation parameter appropriately chosen to accelerate convergence.

6.0 CONCURRENT TESTING OF SYSTOLIC ARRAY PROCESSORS

Utilization of any functional device in realizing important system features ultimately leads to questions regarding reliability and maintainability properties. In this section interesting methods for externally testing systolic arrays for proper operation will be considered. It is not practical to consider reliability features here; therefore, only issues related to maintainability, namely testability, will be considered. Only external methods for testing will be explored.

Consider the systolic array for performing a matrix-vector product originally proposed by Kung (1978). Given the way in which the matrix rows pass through the processor array, a rather simple external test for proper operation of the array would be to augment the given matrix by adding two check rows - one at the top and another at the bottom. This is illustrated in Fig. 11 where the two additional rows must be identical in order to facilitate the check. Note from Fig. 11 that if no $x_i = 0$ and no augmentation element is zero, each processor will be checked in the process of performing the matrix-vector product. The test is very simple since it requires only that y_1 be compared for equality with y_{n+2} .

Two additional processors are required to realize this test. It is interesting to examine the cost required to implement this check in terms of added hardware and algorithm execution time. Let S represent the hardware required to realize a processor in the array and t denote the time interval required for each shift in passing the matrix through the processor. For an $n \times n$ dense matrix and using the product $S \cdot$ (computation time) as a measure of resources used, then the efficiency is given by:

$$\eta = \frac{(S \cdot 2nt) \text{ without test}}{[S(2n+2)t] \text{ with test}}$$

$$\eta \approx 1 - 2/n$$

For n large, it follows that this is a very efficient test in terms of required resources.

With respect to test effectiveness, however, questions follow with regard to fault coverage. If x is known to be dense and the augmentation does not use zero elements, the test will be good for detecting hard failures. However, transient failures represent a problem for this approach.

The test method just described can be applied to matrix-matrix processors, although comparison of more quantities is required. It also follows that this approach is applicable to the PRT transform. Note for this case from Fig. 11, however, that for about n time steps no checks on the computation are performed. This can be overcome by additional augmentations, appropriately interspersed, in the original matrix.

7.0 CONCLUSION

Systolic arrays represent a potentially important means for implementing computations involving large-scale matrices. The realization of a general matrix-oriented computing capability that is founded upon a few standard modules using VLSI technology is appealing. However, as emphasized by Kung (1978), minimization of wiring requirements (communication costs) is a central problem in this technology. The PRT transform introduced in this paper can significantly reduce these costs for some problems. Of particular importance is the fact that these savings can be realized in some cases without increasing algorithm time.

It has been shown that for $n \times n$ banded matrices the PRT-based systolic array and that originally proposed by Kung (1978) are complimentary in the sense that when one is efficient, the other form tends toward lower efficiency. The PRT transform does not alter the original systolic array hardware definition. The time-ordered outputs are invariant under this transform - the only changes appearing in the order of accumulation of intermediate values before they are output at the array port(s).

Solution of linear, simultaneous equations by iteration methods using systolic arrays results in an interesting interpretation of the PRT transform. The PRT or some equivalent transform appears necessary in order to apply systolic arrays to Gauss-Seidel's method or to the SOR method.

A simple, efficient - though somewhat limited - testing technique was introduced for performing external concurrent tests on systolic arrays. This topic, as well as the others considered in this paper, is worthy of further study.

REFERENCES

- Dahlquist, G. and A. Bjorck, 1974, Numerical Methods, Prentice Hall, Inc., New Jersey.
- Kung, H.T. and C.E. Leiserson, (1978), "Systolic Arrays for (VLSI)," Carnegie-Mellon University, Pittsburgh, Pa., (last revised December, 1978).
- Mead, C., and L. Conway, (1980), Introduction to VLSI Systems, Addison-Wesley, Reading, Massachusetts.
- Sameh, A.H., 1977, "Numerical Parallel Algorithms--A Survey," in High Speed Computer and Algorithm Organization, editors Kuck, Lawrie, and Sameh, Academic Press, NY.
- Speiser, J.M. and H.J. Whitehouse, (1980), "Architectures for Real-Time Matrix Operations," GOMAC (Government Microcircuit Applications Conference) Digest of Papers, Houston, Texas.

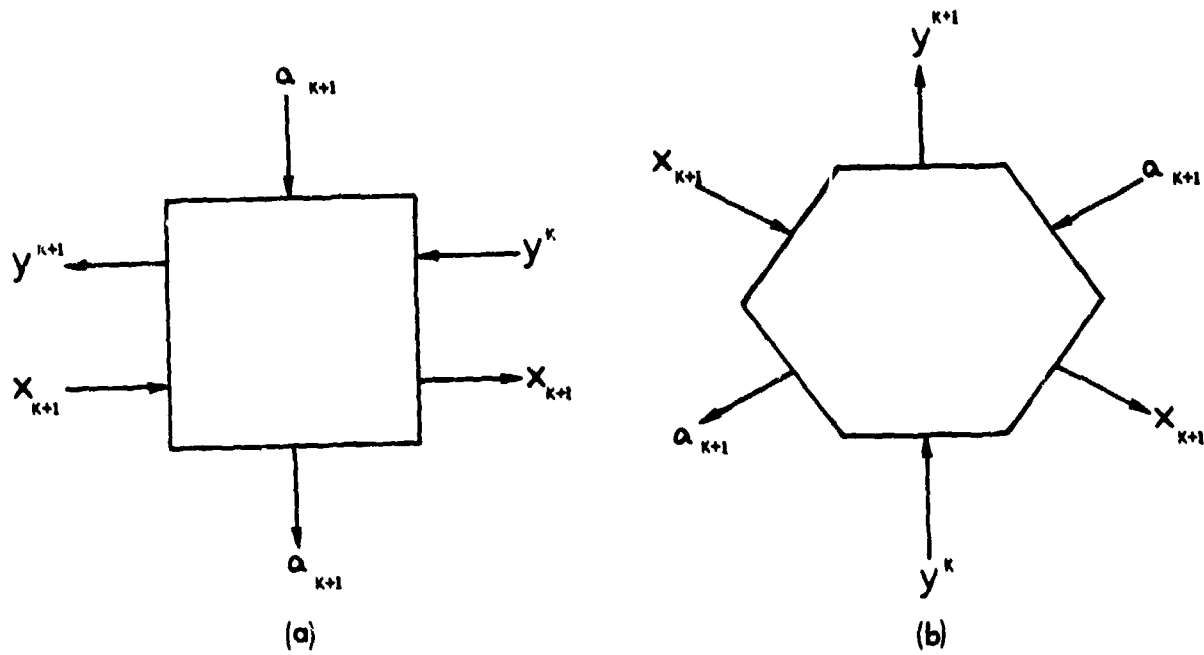


Fig. 1. Two Types of Inner Product Step Processors: (a) Type 1. (b) Type 2.

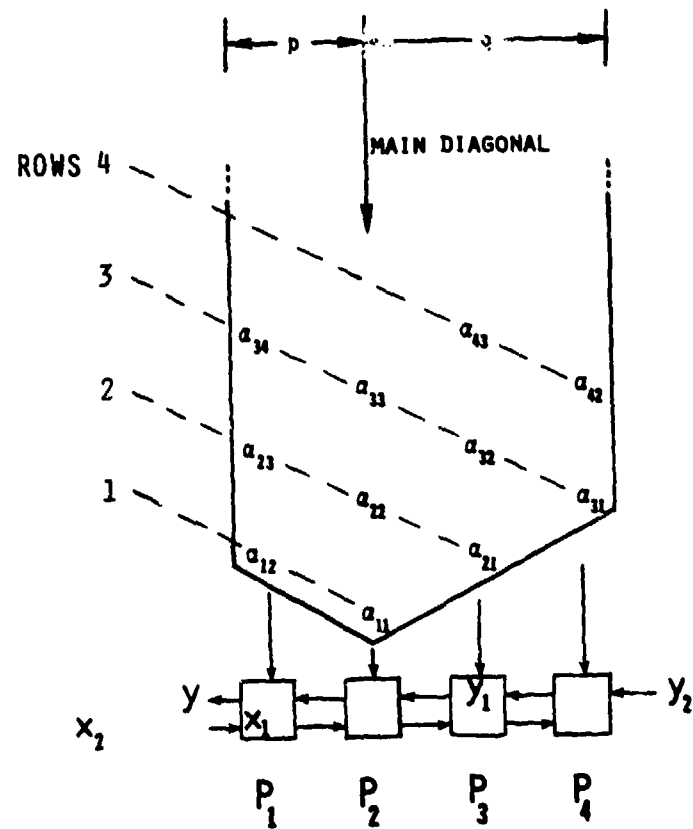


Fig. 2. Systolic Array Processor Configured to Form Matrix-Vector Product $y = Ax$.

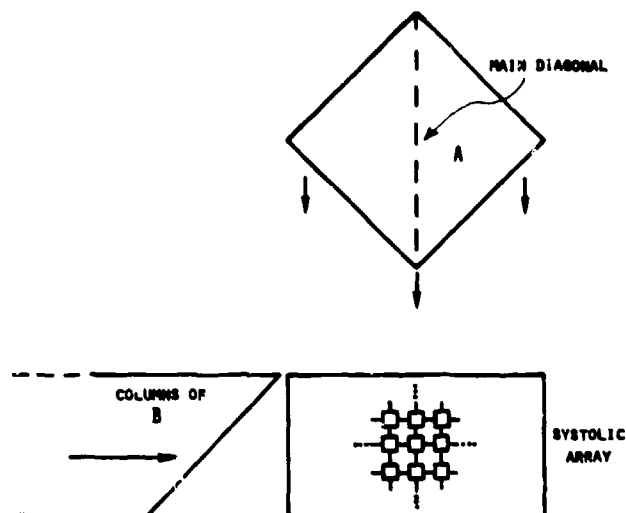


Fig. 3(a). Orthogonally Connected Systolic Array to Compute Matrix Product $C = AB$.

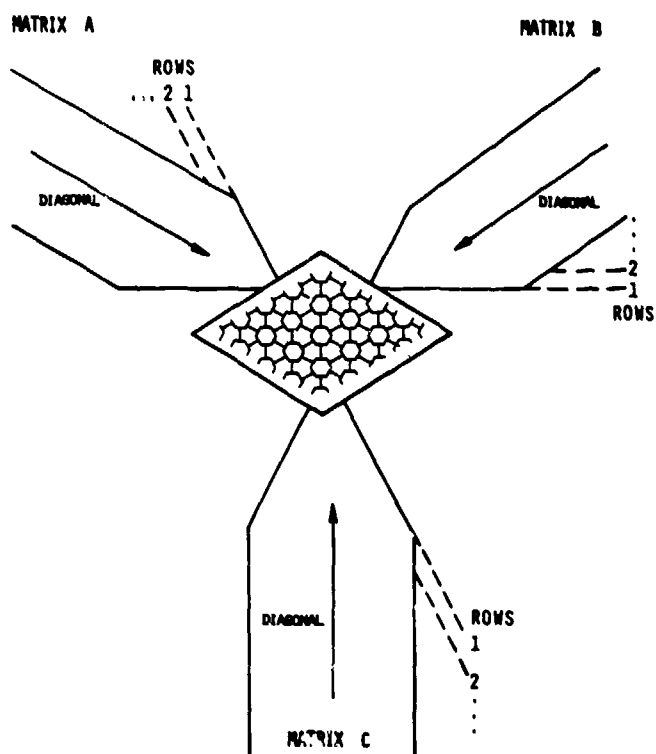


Fig. 3(b). Hexagonally Connected Systolic Array to Compute Matrix Product $C = AB$.

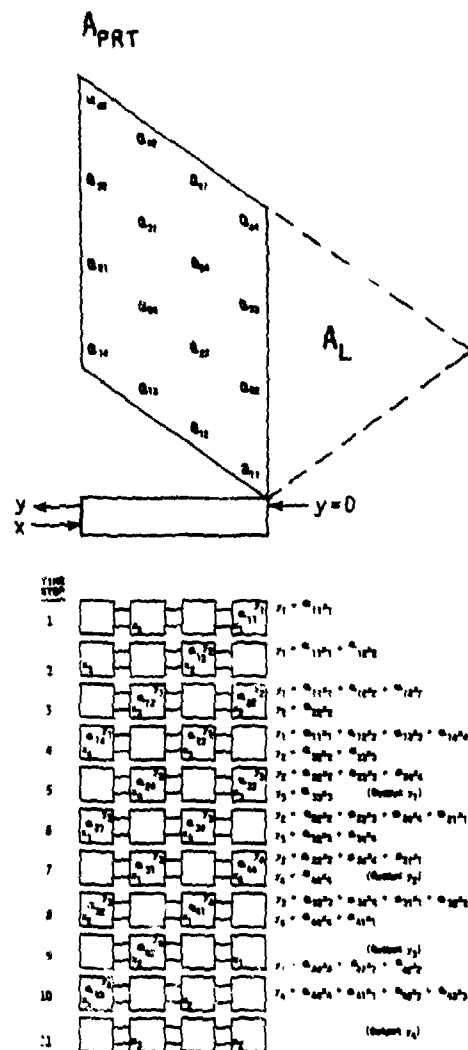
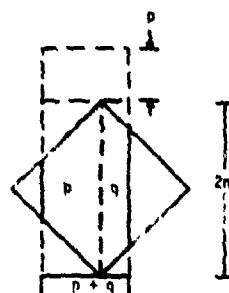


Fig. 4. Detailed Example of PRT Transform and Linearly Connected Systolic Array for Evaluating Matrix-Vector Product $y = Ax$.



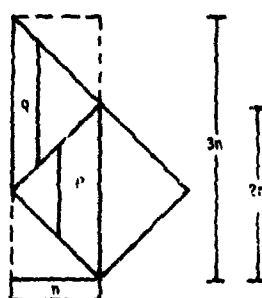
ORIGINAL FORM

$$\eta = \frac{2 - (1-x)^2 - (1-y)^2}{(2+x)(x+y)}$$

$i = 2n + p$, without immediate unloading

$T = 2n$, with immediate unloading

$S = p + q$



ALTERNATE FORM

$$\eta = \frac{2 - (1-x)^2 - (1-y)^2}{3}$$

$T = 3n$

$S = n$

$$\eta = \frac{\text{Active Area}}{\text{Total Area}}$$

$$y = p/n$$

$$x = q/n$$

ACTIVE AREA

TOTAL AREA

Fig. 5. Performance Comparison of Original with Alternate Systolic Array for Matrix-Vector Problem with Square Banded Matrix.

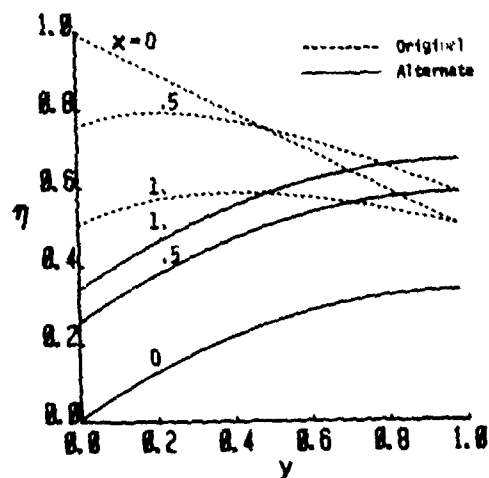


Fig. 6. Processor Utilization Efficiency Versus Matrix Bandwidth Assuming Immediate Unloading Capability.

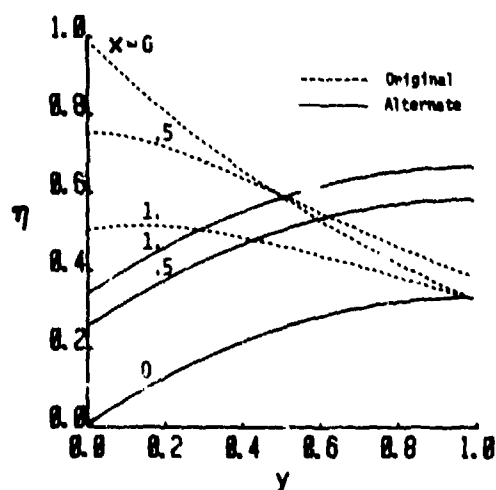


Fig. 7. Processor Utilization Efficiency Versus Matrix Bandwidth Without Immediate Unloading Capability.

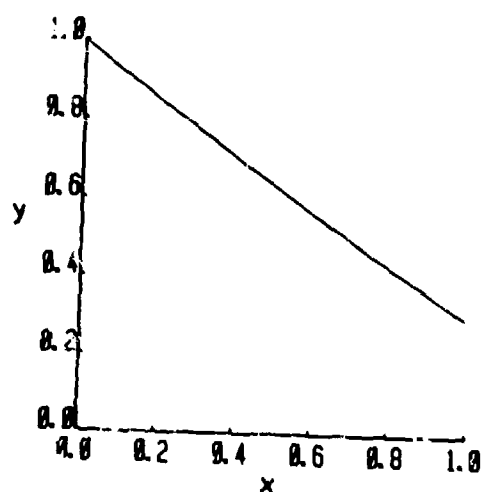


Fig. 8. Values of x, y Which Satisfy the Equality $(ST)_{orig} = (ST)_{alt}$ Without Immediate Unloading Capability.

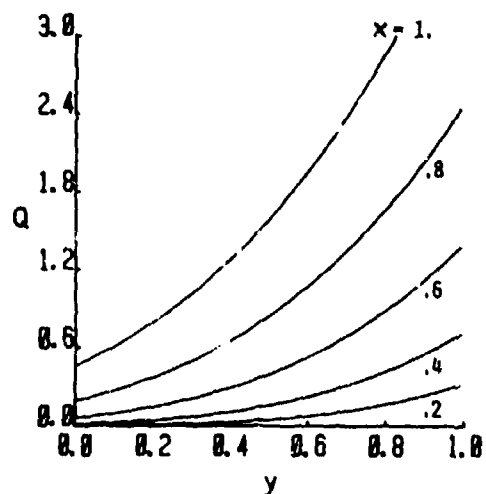


Fig. 9. Figure of Merit $Q = F_{alt}/F_{orig}$ Without Immediate Unloading Capability.

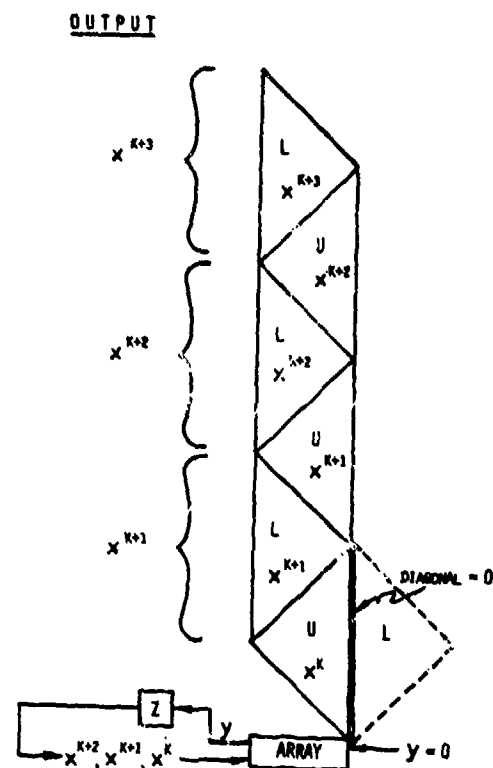


Fig. 10. Three Iterations of Gauss-Seidel's Method on a Systolic Array with External Computations Performed in Block Labeled Z (Initial Estimate $= x^k$).

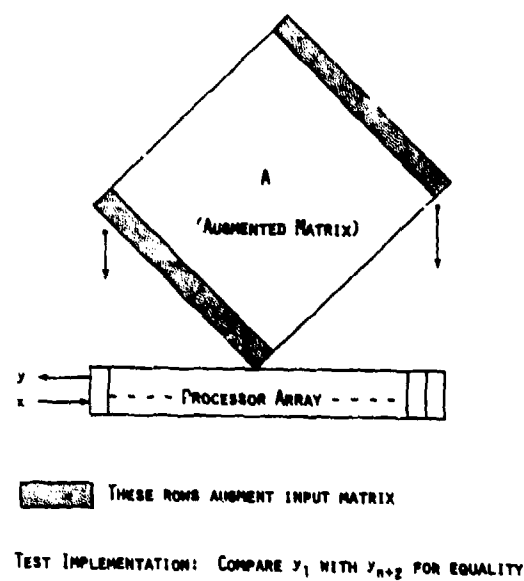


Fig. 11. Concurrent Testing of Systolic Array Matrix-Vector Processor by Augmentation Method.

ECONOMIC CONSIDERATIONS FOR REAL-TIME NAVAL AIRCRAFT/ AVIONIC DISTRIBUTED COMPUTER CONTROL SYSTEMS

BERNARD A. ZEMPOLICH
DEPUTY TECHNOLOGY ADMINISTRATOR FOR COMMAND, CONTROL AND GUIDANCE
RESEARCH AND TECHNOLOGY GROUP
NAVAL AIR SYSTEMS COMMAND, WASHINGTON, D. C.
20361

SUMMARY

Using naval aircraft/avionic systems as examples, economic considerations for Distributed Computer Control Systems (DCCS) are discussed. Centralized, distributed and federated processing architectures are used as the primary set of systems alternatives from which economic factors are developed. Technical, schedule and financial risks for the system architectures are presented. Standardization of computer hardware and software is examined from the economic viewpoint and other related risk factors. The economic impact of subsequent logistic support for standardized computer hardware and software versus non-standard products is identified. System considerations such as reliability, maintainability, availability, built-in-test, fault tolerance, and redundancy are examined from the standpoint of resources available to design and develop the DCCS, and also from the viewpoint of economic impact of failure of the DCCS to perform as expected. The economic impact of external factors such as the rate of technology advancement, technology independence, limited production runs, and the general lack of economic leverage upon the market are examined and related to the life-cycle support requirements of the DCCS.

1. INTRODUCTION

The inherent nature of microelectronic circuitry is that it lends itself to digital design techniques with ease. This attribute, coupled with the microprocessor, provides a powerful design capability to developers of aircraft/avionic systems. Today, powerful microcomputers can be embedded directly into each aircraft/avionic subsystem with little if any impact on weight and volume. With this capability, top-down structured aircraft/avionic systems based on distributed processing and architectures, have become implementable and cost effective.

This paper addresses economic considerations associated with the design of real-time aircraft/avionic Distributed Computer Control Systems (DCCS) for future Naval aviation Avionic System Architectures. The aircraft as a DCCS is examined based on stated aircraft mission and avionic system requirements. DCCS design options and alternatives such as physical implementations and alternative processing architectures, standardization, commonality, reliability, maintainability, and availability are analyzed from the economic viewpoint.

In addition to the options and alternatives available to the developing activity, there are many external design factors which affect the design for an Avionic System Architecture over which the developer has little or no control. Among them are the rate of technology advancement, technology dependence and independence, and the general lack of economic leverage by the developers over the products of the solid state industry. These factors are addressed from the viewpoint that management must be aware of their potential impact on the design of a DCCS.

2. ECONOMIC CONSIDERATIONS

2.1 DCCS SYSTEM DESIGN OPTIONS AND ALTERNATIVES

As with most engineering efforts, the design of an aircraft DCCS allows the developer to exercise a number of options, all which have inter-related technical, schedule, and economic (cost) risk. DCCS design options and alternatives generally fall into two categories--those factors over which the designer has direct control and those factors over which there is little or no control by the developing activity. DCCS considerations over which the developer has control include: physical implementations; alternative processing architectures; standardization and commonality; and reliability, maintainability, and availability. Factors over which the developer has little or no control are all economic in nature. Among these external considerations which impact the design of a DCCS are the following: the rate of technology advancement, technology independence, and lack of economic leverage in the marketplace.

2.2 PHYSICAL IMPLEMENTATIONS

As stated previously, once the primary mission for an aircraft is established, the Avionic System Architecture can be decomposed into functional requirements. In a similar fashion, subsystems can be partitioned into various physical implementations. There are three basic equipment physical implementation alternatives: the "black box" approach, the form, fit and function (3F) approach, and the integrated technologies concept.

With the black box approach, all equipment procurements over the life-cycle of the aircraft are bought to a set of specifications which detail not only the function and form, but also the internal configuration--electronic, electromechanical, and packaging. Once the desired performance of the production units is established, subsequent procurements usually have minimum technical and schedule risk. Quantity of units to be bought per unit of time is the dominant economic factor with procurement of black box implementations of avionic equipments. Multiple suppliers can also be considered a major force in price determination as the competitive atmosphere tends to keep the per unit cost of the equipment down. The assumption here, of course, is that alternate sources have the capability to produce the equipment with no technical and/or cost problems. Lastly, with the black box approach, long-term logistic considerations (which have a great impact on the life-cycle costs of the aircraft) can be established after the equipment reaches production maturity.

A second physical implementation alternative for avionic equipments is that of form, fit and function (3F). With the 3F approach, procurements of equipments are made based on a set of specifications which detail the required physical dimensions as well as the electronic and electromechanical interfaces. The technologies of the assemblies within the unit, on the other hand, are allowed to vary or "float". The economic value of the 3F approach rests mainly with the options open to the supplier in having to meet only the 3F specifications. In essence, the supplier is free to make maximum use of his particular resources, design approaches, and manufacturing facilities. It is normal to expect that there is the potential for cost savings through the use of the 3F approach in that it permits more suppliers to bid. However, there is an economic shortcoming of the 3F approach in that it does not readily lend itself to long-term logistic gains and planning. This shortcoming may be minimized if the alternative supply source were to use components and/or parts already in the customer's inventory.

In both the black box and 3F approach, each avionic unit performs a fixed specific function. At the other end of the spectrum, the Avionic System Architecture can be partitioned along the lines of integrated technologies in which functions are performed by generic task areas such as data processing, communications, navigation, or controls and displays. In this instance, advanced technologies are used in an integrated fashion such that any one given part of the subsystem is capable of performing different functions at different times. Specifically, with the integrated technologies implementation, the functional elements are all electronically reconfigurable. While this concept has considerable potential performance and economic merit, it has yet to be fully exploited in avionic applications, and thus the risks are not yet well established.

Regardless of the alternative chosen, the selection of the physical implementations of aircraft/avionic equipment(s) is a fundamental design decision which has major technical and management impact during the development phase as well as during the operational life of the aircraft. For this decision dictates life-cycle logistic support approaches for the system such as depot repair, module "throw-away" concepts, or factory repair and maintenance.

If the decision regarding which physical implementation alternatives should be selected could be made on the considerations just addressed, the choice is reduced solely to a comparison of risks. Unfortunately, the choice is also dependent to a large degree on the proposed aircraft installation. Specifically, is the installation of the DCCS to be made in an existing operational aircraft as opposed to an installation in a new airframe? With a new airframe, the weight, volume, and location of the equipment is normally determined concurrently with the development of the aircraft, thus there is a degree of design latitude allowed in the physical integration of the aircraft/avionic subsystem. On the other hand, with an existing airframe, there are a number of significant restrictions on the installation of a newly designed DCCS because of the need to conform to existing physical conditions.

The importance of installation options cannot be overstated. Restrictions that may have to be faced when installing equipment into existing aircraft may very well prevent an optimal combination of airframe and on-board aircraft/avionic subsystems from a logistic viewpoint. Needless to say, logistics considerations are for all practical purposes economic considerations, and if experience to date is any measure, the costs of lifetime logistical support far exceeds the non-recurring development costs.

2.3 ALTERNATIVE PROCESSING ARCHITECTURES

The modern aircraft/avionic DCCS will be required to handle a wide variety of tasks ranging from complex, high speed signal processing to simple input/output formatting and control. Additionally, fault-tolerance concepts demand that many of the processing elements within the DCCS be capable of reprogramming during the operational mission. The overall processing architecture must therefore support the synchronization, control, configuration, reconfiguration, and fault-detection of all processors in the DCCS. Furthermore, to minimize architectural problems, both the hardware and the software must be functionally partitioned in such a manner that the interface complexity is manageable, and the design and implementation of each unit processor is maintained in as independent a manner as is possible.

There exists a variety of processing architectures which can be utilized to design an aircraft/avionic DCCS with the performance capabilities just identified. It should be noted, however, that each alternative has attached to its use a unique set of technical, schedule, and financial risk factors. Figure 1, Processing Architecture Alternative Comparison, lists a number of available processing architecture options and identifies the associated risk factors. Risks are stated in low, medium, and high terms because there does not exist a statistical data base from which precise numerical values can be derived.

Unfortunately, the procedure for selecting a specific processing architecture is not solely a matter of looking at the risk factors inherent in the individual architectures and determining what is an acceptable composite level of overall risk to the developer. For example, the Avionic System Architecture Considerations identified in Table 1 also weigh heavily upon the decision concerning which processing architecture is "best" for a specific application. The necessity for having to take into consideration both the processing architecture alternatives as well as other Avionic System Architecture factors provides the developing activity with a myriad number of possible combinations from which to choose during the design of the DCCS. The technical management task required to separate these combinations into a set of hierarchically structured options based upon a well understood set of selection criteria is complex unto itself.

Because of the large number of interrelated factors which affect the selection of a processing configuration for a specific Avionic System Architecture and the lack of a historical cost data base, one can only address in general terms the economic considerations of the various processing alternatives. Even though economic considerations can only be addressed in general terms they should not be interpreted as being either superficial, lacking in importance, nor restricted to only one architectural choice. For even as incomplete as is the cost data at this point in time, trends can be drawn from experiences with the individual requirements of current aircraft/avionic systems. Examples of considerations which have significant impact upon the life-cycle cost of DCCS and require detail management attention by the

developing activity during the project planning phase are: degree of system integration, degree of partitioning of the system, software, firmware, and hardware trade-offs, and software cost/complexity.

2.4 DEGREE OF SYSTEM INTEGRATION

This issue addresses the degree of total system integration of the Avionic System Architecture. For example, should the categories or groups of subsystems identified earlier be placed on a single high-speed data bus or should each group have its own dedicated data bus to perform functions particular to the individual grouping of subsystems. A specific example of the dedicated data bus would be to keep all vehicle-related subsystems segregated for safety-of-flight reasons. It can be anticipated that if there is one high-speed data bus throughout the aircraft, then the complexity of controlling the data bus and performing real-time executive and interrupt functions would be increased dramatically. In turn, software-related costs (design, test and documentation) would increase significantly, if not proportionately with the degree of integration. This conclusion is based on the fact that cost experience (in terms of dollars per instruction) with operationally deployed aircraft systems to date has shown that the real-time executive and I/O routines are much higher than application programs and test and diagnostic routines.

2.5 DEGREE OF PARTITIONING OF THE SYSTEM

As stated earlier, future aircraft DCCS's must be designed using a structured process of decomposition into software, firmware, and hardware processing modules. In future aircraft, the degree of distribution (partitioning) of computing, control, and conversion functions, will be dependent on the availability of inexpensive and physically diminutive hardware elements--namely microprocessors and microcomputers. It should be noted, however, that while the use of a central computer complex to provide functional digital control of an aircraft has deficiencies due to the multiplicity of tasks which must be performed in one machine, the DCCS has yet to face the same problems while performing similar tasks with as many as up to 150 to 200 (micros) machines.

2.6 SOFTWARE, HARDWARE, AND FIRMWARE TRADE-OFFS

The programmable digital computer allows in-service functional change without impacting the associated hardware, except where additional memory is required. With the recent introduction of firmware, the "best of two worlds" is available. Furthermore, the options for committal of functions to firmware implementation as opposed to software is unbounded in number. Key to any decision-making process as to whether or not to put a function into firmware is when should one freeze the software program design and how often, if ever, is the program going to be required to be changed throughout the operational lifetime of the system. Any misjudgement on the proper timing for freezing the program into firmware and miscalculation on the number of times that the firmware will require subsequent change, will result in major increases in development and support costs.

2.7 SOFTWARE COST/COMPLEXITY

In the centralized processing architecture, the cost and complexity of Applications/Control and Input/Output programming rises exponentially as the throughput and memory of the centralized computer approaches its maximum (see Fig. 2). On the other hand, with the distributed processing architecture, the Cost/Complexity at near zero percent (0%) distribution is the same as one hundred percent (100%) utilization of a centralized computer system. As the degree of distribution (i.e., partitioning) is increased, each application software module becomes more independent and has less effect on the execution of the total on-board system processing (program). The I/O program, however, becomes more complex since more processing elements (micros) must be interfaced via the data bus structure. The data availability and I/O control becomes the dominant factor, ultimately following the I/O program curve of the centralized computer system in rising Cost/Complexity (see Fig. 3). The sum of the software trends indicates that there is probably a point at which partitioning may be optimal. As is self-evident from Fig. 3, at either end of the percentage distribution spectrum, the worst of both situations may exist.

2.8 STANDARDIZATION AND COMMONALITY

It is the author's opinion that no other area of the data processing field is more complex in scope and controversial in nature than the area of standardization. Many professionals in the field of data processing do not agree that standardization has both technical and cost merit. This lack of consensus on the worth of standardization is due to the naturally opposing views of computer system users and the developers of computer systems. For the user views standardization as a means of management control of development risks and system life-cycle cost control, while the developer and designer, on the other hand, views standardization requirements as an unnecessary restriction on technical creativity. Many developers also counter the user's position that proliferation of computer equipment and software is a major life-cycle cost burden with the claim that given design freedom during the development phase of a new system, they would introduce new technologies which would be cost-effective as well as having increased performance capability over existing operational systems. Unfortunately, there is a tendency amongst proponents of this development philosophy not to mention that new designs also give rise to normal self-vested interests, such as increased profits and keeping the in-house design teams current with involvement in emerging technologies and techniques. These two diametrically opposed positions will never change in this author's opinion, as the developer normally will only address the technical and financial aspects of the specific systems he is developing; while the user, on the other hand, is concerned with standardization as applied to multiple system applications. Additionally, there is another dimension to the standardization issue which often is not considered in any discussion of computer systems standards.

Specifically, the question is at what point or level does one standardize? For example, one could standardize at the Instruction Set Architecture (ISA) level while allowing the designer to incorporate the latest technologies, change the physical and electrical characteristics (e.g., overall dimensions, the internal mechanical structure of the machine, and cooling and primary power requirements).

Table 2, "Standardization Options", lists a number of possible standards which the user and/or the developer of aircraft avionic equipment could adopt. Several or many of these options could be combined to form an all-encompassing single standard depending on the financial resources available, maintainability/support approaches, and the end operational use of the system(s). However, the more these standardization options are molded into one single standard, the greater will be the negative reaction of the developer, as stated earlier.

TABLE 2 STANDARDIZATION OPTIONS

- Languages
 - Preprocessor (POL)
 - Compiler (HOL)
 - Assembler (MOL)
- Instruction Set Architecture (ISA)
 - Single Instruction Set
 - Modular Instruction Set
 - Extensible Instruction Set
- System-Level Interconnection Schemes
 - Bus
 - Loop
 - Network
 - Bus Interface Unit
- System-Level Protocol
 - User Module to Operating System
 - Operating System to Hardware
- Physical Interface
 - Pin Compatible
 - Plug Compatible
- Physical Implementation
 - Black Box
 - Form, Fit, Function
 - Standard Module
 - Micro-chip Set

Of all the Standardization Options listed in Table 2, adoption of an Instruction Set Architecture (ISA) as a standard offers the greatest economic return on investment to the customer. This is assuming that the ISA selected as a standard has an established user and support software base.

If one were to address the standardization issue solely on the basis of generalized hardware and language (HOL) alternatives, then a matrix of comparative risks can be defined. Figure 4, Hardware Standards, shows the technical, schedule, and financial risks for various hardware alternatives. It should be noted that high and medium/high risk factors have been assigned to the Strict Processor and Microprocessor Standards because of: (1), the lack of experience with building DCCS's for aircraft/avionic systems applications; and (2), it is not clear at this point that a single, cost-effective microprocessor can be established as a standard for all applications throughout an Avionic System Architecture.

The key issue relative to establishing a microprocessor as a standard piece of hardware is at what point does one not enforce standardization. For example, is every application which calls for a microprocessor whose word length is less than 16 bits subject to the standard? Or, is there a minimum memory size below which the microprocessor would be excluded from standardization considerations? These decisions, while seemingly inconsequential, do have a significant impact on the design of the system and development costs.

Many individuals have postulated that microprocessors will decrease the cost of computer hardware to the point at which it is an insignificant factor on future developments of DCCS's. This claim has yet to be proven. Unfortunately, the rising costs of both applications and support software have lent credibility to the position that the cost for microprocessors are no longer of relative importance in system life-cycle cost considerations.

Regardless of the availability of comparatively low-cost microprocessors and microcomputers, the high cost of software development and maintenance has given considerable support to the utilization of HOL's and, in particular, a single HOL wherever possible. Figure 5 indicates that assembly level coding is definitely more costly than that of using HOL(s). There are two major reasons for this cost differential: (1), there is a need for the programmer to know the particular Instruction Set Architecture of the target machine(s); and (2), in most cases assembly level code is used mainly for very difficult program tasks such as: input/output, operating systems, and executive control of real-time systems. In each of these instances the programmer must work with "tight" coding requirements.

Within the context of this paper, commonality is defined as the utilization of equipment(s) of parts thereof, in multiple operational applications. For example, many aircraft cockpit controls and displays could be common within a single "family" of aircraft types. Each aircraft, however, would have a specific set of cockpit controls and displays tailored to its own particular operational need. Across all aircraft within the family, the controls and displays would perform common functions. The equipment itself need not be standard items to be considered within the context of commonality as the term is used herein. (See Figure 6.)

The potential for major cost-savings does not exist with the utilization of common equipment as it does with standard equipment because of the specific tailoring or uniqueness of the equipment to each application. On the other hand, when the developer applies commonality concepts effectively, there is a great potential for significant cost-avoidance. For example, specific display components, bulk memories, algorithms, etc., can be applied across all applications. In doing so, the developer avoids those costs associated with developing totally unique equipment designs for each installation.

2.9 RELIABILITY, MAINTAINABILITY, AND AVAILABILITY (RMA)

In simplistic terms, aircraft/avionic systems are designed to meet pre-established levels of reliability so as to be available for operational use for given time periods prior to a failure occurring which would require a maintenance action to be taken. When the reliability levels are not achieved, the equipment is not available and additional maintenance actions have to be taken. This cause and effect situation is a major contribution to operational support costs. In the author's opinion, it is highly unlikely that with the current degree of technical sophistication of aircraft/avionic equipment that these costs will decrease in the near future. Furthermore, unless new Avionic System Architectures are developed and designed as described earlier, the current RMA problems will remain.

It should be emphasized that using a DCCS as the basis for a future Avionic System Architecture will not of itself negate the current RMA problems, however, if the system is designed in a structured manner, it can include many features which would assist in reducing RMA shortcomings exhibited by current operational systems. Key features which will have a major impact in improvement of RMA factors and a corresponding reduction in life-cycle operational costs are: fault-tolerant, redundancy, and reconfigurability.

The capability to incorporate fault-tolerant, redundancy, and reconfigurability techniques and concepts into a DCCS is based primarily on the availability of relatively inexpensive microprocessors. Given that these microprocessors will be available, the major question remaining is at what level does the developer insert these concepts into the design of the DCCS. For these concepts can be applied either on a system-wide basis, or at any of the subsystem or functional grouping levels. Furthermore, with the coming of age of the reconfigurable memory, one can now have increased availability at the component level.

The coupling of fault-tolerant, redundancy, and reconfigurability with automated fault-detection and isolation also offers management a vehicle for minimizing RMA life-cycle cost for future DCCS's. Unfortunately, the expected theoretical improvements in the RMA values have yet to be fully proven out in actual practice over a substantial period of operational time. While there is no reason to believe that the potential gains cannot be achieved, there is an area of concern (mentioned earlier) that should be addressed during the development of the Avionic System Architecture--namely that of the actual amount of distribution of computing resources throughout the system and its impact upon the associated software.

The complexity of the software associated with a DCCS is going to be a major challenge by itself. There are many problems yet to be faced with an aircraft/avionic DCCS which may contain over 150 microprocessors throughout the aircraft. Additionally, there could be hidden costs because of unforeseen needs for performing extensive test and evaluation of such a system. Hopefully, sufficient software verification and validation techniques will be available to insure that the developer can adequately separate proving the quality of the software from the quality of the DCCS to function adequately as an integrated network of computer resources.

3. EXTERNAL FACTORS IMPACTING DCCS DEVELOPMENTS

3.1 EXTERNAL FACTORS

The question that developers of an aircraft/avionic DCCS must ask themselves before starting out on a new design is what degree of control do they have over their final design. Unfortunately, the dynamics of the microelectronics industry as mirrored by the microprocessor/microcomputer marketplace presently defy the providing of reasonably precise answers to the question. At best, one can only hope that the impact upon DCCS development efforts and related life-cycle consideration of the Avionic System Architecture are minimized through the recognition of external factors during the planning phase of the project. The following external factors are identified as having a major impact upon the DCCS design and development and thus should be addressed during the planning phase of the project: the rate of technology advancement, technology dependence/independence, limited production runs as a function of time and lack of leverage upon the market, technology transfer and insertion, and the vertical structure of certain corporations.

3.2 TECHNOLOGY ADVANCEMENT

It is almost inconceivable that the technological inventiveness of the solid state electronics industry is such that new products become obsolete almost immediately after introduction into the marketplace. Breakthroughs in such areas as materials, manufacturing processes, computer aided design, architectures and packaging are made almost daily. Furthermore, it is highly unlikely that in the near future there will be any slow-down in new performance capabilities being introduced in the microprocessor/microcomputer marketplace. If anything, there will be a continued explosion of new applications as the prices of these machines (micros) decrease as a function of time.

All other design factors being equal, advancements in the solid state electronics field are not necessarily detrimental to the aircraft DCCS developer. Desired system-level capabilities such as redundancy, reconfigurability, and fault-tolerance can now be built into the system economically and contribute to achieving the desired performance goals set for system maintainability, reliability, and availability. On the other hand, these capabilities cannot be logistically supported over the life-cycle of the system DCCS without taking into account the other external factors which impact DCCS developments.

3.3 TECHNOLOGY INDEPENDENCE

In similar fashion to the commercial computer industry expression of "plug-to-plug" compatibility, the phrase "technology independence" has been introduced into the military-industry lexicon. In a manner of speaking, it can be considered a technology level equivalent to the form, fit and function (3F) physical implementation approach addressed earlier. The concept is very simple, that is, by being independent of technology uniqueness one can insert new technologies at given time intervals during the life-cycle of the aircraft/avionic DCCS. The economic return on investment for incorporating this capability into the initial system design is significant. On the other hand, it does demand that there be some level of mechanical packaging standards in order to introduce the new devices and/or components into the existing equipment with minimum impact upon the associated logistic considerations. Assuming that a standard mechanical packaging concept can be established for both the in being and the potential replacement technologies, then there will be a logistic cost avoidance in that the higher level electronic assemblies do not change with the insertion of the new technology.

With regard to software, however, technology independence takes on a number of meanings, all of which depend on the point of view of the developer. For example, applications and support software for a given programmable digital computer could be run, with no changes, on a newer technology machine providing the Instruction Set Architecture and other software program dependent characteristics are taken into consideration during the initial design phase. A second conceptual approach would be to keep the High Order Language (HOL) interface independent of the operational target machine. Lastly, a third approach would be that of using a pre-processor in the software development chain. Specifically, with this approach, one establishes the near-equivalent of a hardware plug-to-plug compatibility by using a pre-processor as a software program translator. In this instance, the firmware is used to provide the software compatibility link.

Regardless of the type of hardware technology used, the concept of software transportability implies unto itself, technology independence. However, unlike hardware technology independence, software transportability of its very nature explicitly implies reusability of software as opposed to the basic concept of plug-to-plug compatibility; namely that of technology insertion through technology invisibility (independence).

It can be generally stated that software transferability offers the developer a basis for cost savings. On the other hand, since the application/program will no doubt be different to a certain degree from functional task to functional task, new compilations will have to be performed in order to insert different application dependent parameters and data. Thus it is perhaps more correct to state that as a minimum, using software transportability concepts in an aircraft/avionic system design there will be a cost avoidance in that both the operational and support programs do not have to be re-created from the initial design stage.

3.4 LIMITED PRODUCTION RUNS

There is not a better method to insure price stability than that of having the advantages that accrue from large scale procurements over a given period of time. In essence, this is the economy of scale factor of classical economic theory. Unfortunately, it is a fact of life that at best there will be limited quantities of aircraft/avionic Digital Computer Control Systems procured by any one development/procurement activity. For example, even if an aircraft manufacturing firm has incorporated DCCS's (utilizing microelectronic chips) into several different aircraft models, the quantities of either commercial or military aircraft coming off the production lines are miniscule compared with the quantities of microelectronic chips currently being procured by both the automotive and toy industries on a per year basis.

It would appear that there are two management alternatives which would overcome the inherent economic shortcomings of limited production runs for military applications of commercial components. The first approach, would be to add onto existing commercial production runs which are expected to produce microelectronic chips over an extended period of time. In this instance, individual procurement of chips for the DCCS would be made part of a standard product line which the solid state electronics firms expect to market to multiple users for into the foreseeable future.

In the second case, the aircraft/avionic systems manufacturer would "front-end" the development costs associated with the design of a given microelectronic chip and only use the solid state electronics firms as a production facility. Thus, the system developer order parts to his specifications and is not dependent upon the microelectronic circuit manufacturers for any initial non-recurring investment in chip design and development costs.

It is essential that an acceptable manufacturing alternative be established prior to production in order to maintain the availability of chips throughout the lifetime of the DCCS or until the chips are replaced by a new technology during the operational phase of the system life-cycle. It is imperative to note that the lack, or shortage, of logistic spare parts destroys any logistic planning performed during the R&D stage of the DCCS and further compounds the subsequent operational problems which range from day-to-day system availability to long term maintainability and reliability.

3.5 LACK OF ECONOMICAL LEVERAGE

Since World War II, the aerospace industry has introduced many advancements in the electronic state-of-the-art into the operational environment. In general, the industry has introduced new technologies because they have had both the performance need as well as the economic leverage to do so. Over the last decade, this preemptive position has been eroded so that presently the aircraft/avionic developers have very little impact upon the technological directions of the solid state electronics industry (based upon a percentage of sales). Neglecting such considerations as global macroeconomics, the changing role of the multi-national firms, and the emergence of a truly international capability to manufacture solid state electronic devices, no single factor has had such a major negative impact upon the economic leverage of the aircraft/avionic firms over the solid state electronics marketplace as that of the coming of age of microelectronic circuitry. That this is so is so ironic in that the aerospace firms first introduced integrated circuits into aircraft/avionic application in the early 1960's.

Since the mid 1960's, the combined sales of aircraft/avionic systems to both the private and public (defense and space) sectors has declined. While decreasing sales volume of aircraft per unit of time has had a profound negative effect upon industry leverage, it has really been the quantum jump in densities of the chips (transistors per unit of area) which has become the dominant factor in changing who, in the private sector, has the economic leverage over the solid-state industry. That this is so should be somewhat self-evident in that the higher density chip development made obsolete the first generation "integrated circuit". It was, for all practical purposes, a single (physical) low cost replacement for hundreds of individually packaged integrated circuits. Thus, in reducing by orders of magnitude the number of chips to be procured, all vestiges of economic power over the solid state marketplace by the aircraft/avionic system developers disappeared.

In retrospect, it is somewhat ironic that in the early 1960's it was the aircraft/avionic industry that was the only group of users that "carried" the then infant microelectronic industry during those days of high-risk integrated circuit venture enterprise. By contrast, today a common 3 to 5 chip microcomputer design serves applications in the aerospace industry, automated factories, medicine, as well as the home entertainment market. On the other hand, projecting into the future, there is the possibility that there may be yet another "role reversal" concerning leverage of the market. Specifically, the use of Very Large Scale Integrated Circuits (VLSIC) in aerospace applications may very well prove to be the key factor in having the microelectronic circuit manufacturers re-tooling to meet once again the unique needs of the aerospace industry. Whether this situation will come to pass has yet to be determined. Until that time, however, aircraft/avionic system developers will have to fit their needs into standard product lines if they do not wish to incur large non-recurring costs for production of customized chips.

3.6 VERTICALLY STRUCTURED CORPORATIONS

Throughout the private sector there are many instances where a corporation is vertically structured--that is where the organization is made up of companies and/or divisions which supply the raw materials, engineering (including R&D), manufacturing, and sales and distribution functions. In essence, the corporation does not go outside of itself for any major aspect of its operations and for all practical purposes is its own supplier of goods and services. The "verticality" of the organizational structure is derived from the nature of the manufacturing process whereby a unit of the corporation builds upon the output of another part of the organization. The management and cost advantages of this situation whereby availability of materials, scheduling, and commitment to corporate goals are all self-contained and controlled needs no further amplification.

With the advent of the transistor, many firms added a solid state technology division (as a separate profit and loss center) to the corporate organization. Except in certain instances, the majority of these solid state technology plants manufactured parts for the general commercial marketplace with no objective of serving internal corporate needs for devices such as transistors. In the author's opinion, the subsequent introduction of the microelectronic chip initiated the push for many aircraft/avionic equipment manufacturers to also take corporate action to change to a vertical organizational structure. For the microelectronic chip took away many design prerogatives from the developers and effectively made the solid state electronics manufacturing firm design competitor, albeit at the very low end of the design process. However, as the techniques for manufacturing microelectronic chips matured, and the industry introduced medium and large scale integrated circuits, the impact upon the classical design freedom of the aircraft/avionic equipment firms became fairly significant as the chips began to contain more and more of the individual circuits previously developed as physically separate designs.

To counter the growing impact of the external factors addressed earlier and the inroads that advanced microelectronic circuitry was making upon their traditional development efforts and organizational makeup, many aerospace firms changed their corporate structures to a vertically-oriented one. What many of these corporations did within the past decade was to create an in-house solid state electronics and technology organization with the prime customer being the corporation itself. The capabilities of these in-house facilities are, as could be expected, as sophisticated and advanced as many of those in California's so-called "silicon valley".

It is premature to state that the vertically-oriented aerospace firm will provide a management approach to overcoming the negative aspects of external factors such as technology advancement and independence, limited production runs, and the general lack of economic leverage over the industry. An exception, of course, is the case where the aerospace company provides chips to other divisions in the organization in bulk quantities.

In general it appears that the creation of an in-house solid state manufacturing facility is a questionable long-term cost-effective solution to the problem. Specifically, the economic law of supply and demand will become a dominant factor relative to the final solution. That is, if the number of firms having in-house solid-state technology R&D and manufacturing facilities increases unabatedly with time, then it follows that in turn, the aerospace firms will become contributors to the herein defined technology and manufacturing external factors over which they currently have little control. It is also not unrealistic to envision

that with time, the aerospace firms will also become suppliers of microelectronic circuitry to the marketplace and thus eventually become competitors with today's solid state electronics firms. To use the cliché, the solution becomes part of the problem.

4. CONCLUSIONS

There are several conclusions that may be reached relative to economic considerations for future Naval aircraft/avionic Real-Time, Distributed Computer Control Systems. The primary conclusion is that designers/developers will have very little economic leverage over the microelectronics industry with the current low rates of production of aircraft and related avionic systems. What follows from this lack of economic control is questionable future enforcement of standardization and commonality requirements. On the other hand, if there is an economy of scale due to a large quantity buy over an extended period of time, then there will accrue to the customer the expected savings in development and support costs. However, with the rapidity of technological change in the solid-state electronics industry, it is becoming more and more self-evident that to fully obtain the economic benefits of standardization and commonality, technology independence over the life-time of the aircraft/avionic system must be maintained.

ARCHITECTURE ALTERNATIVES	RISK		
	TECHNICAL	SCHEDULE	FINANCIAL
(1) DEDICATED SUBSYSTEM PROCESSORS	LOW/MEDIUM	LOW/MEDIUM	LOW/MEDIUM
(2) REDUNDANT DEDICATED SUBSYSTEM PROCESSORS WITH LOCAL BUSES	MEDIUM (WEIGHT)	LOW	LOW/MEDIUM
(3) REDUNDANT DEDICATED SUBSYSTEM PROCESSORS	MEDIUM/HIGH (WEIGHT)	LOW	LOW
(4) REGIONAL GROUPS OF SUBSYSTEMS	MEDIUM	LOW/MEDIUM	LOW/MEDIUM
(5) CENTRAL PROCESSORS	MINIMUM	MEDIUM (INTERFACING)	HIGH (INTERFACING & SUPPORT)
(6) MULTIPROCESSORS	HIGH (SOFTWARE & BUS PROBLEMS)	MEDIUM/HIGH	MEDIUM/HIGH

FIGURE 1

ARCHITECTURE ALTERNATIVE COMPARISON

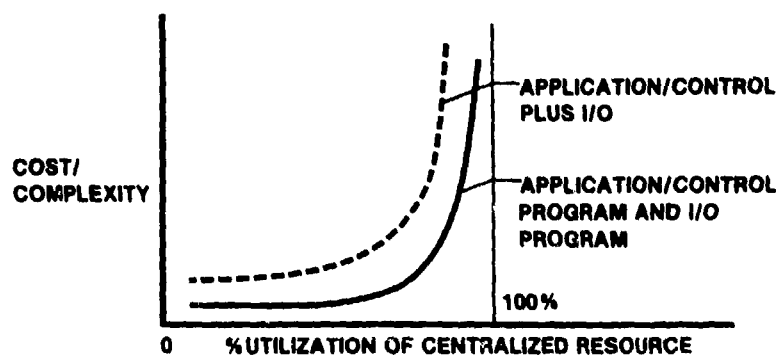
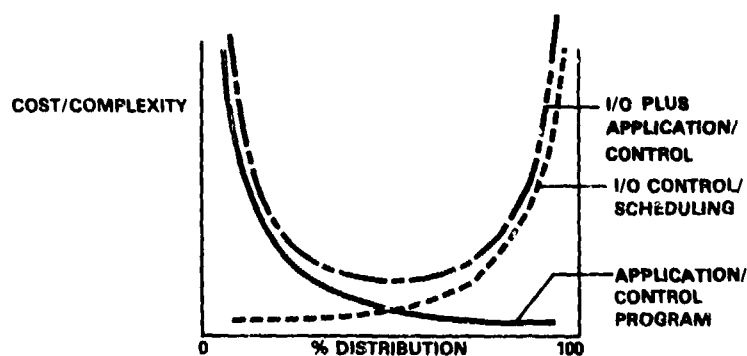


FIGURE 2

CENTRALIZED PROCESSING ARCHITECTURE SOFTWARE COST/COMPLEXITY



- THE SUM OF THE TWO SOFTWARE TRENDS INDICATES A POINT OF DISTRIBUTION WHICH MAY BE OPTIMUM. FURTHER, AT EITHER END OF THE DISTRIBUTION SPECTRUM THE WORST OF BOTH WORLDS MAY EXIST!

FIGURE 3

DISTRIBUTED SYSTEM TRADEOFFS

HARDWARE ALTERNATIVES	RISK		
	TECHNICAL	SCHEDULE	FINANCIAL
STRICT PROCESSOR AND MICROPROCESSOR STANDARDS	HIGH	MEDIUM	MEDIUM/HIGH
PROCESSOR STANDARDS ONLY	LOW/MEDIUM	LOW	MEDIUM/HIGH
NO STANDARDS	MINIMUM	HIGH	HIGH

FIGURE 4
HARDWARE STANDARDS

LANGUAGE ALTERNATIVES	RISK		
	TECHNICAL	SCHEDULE	FINANCIAL
SINGLE HIGH LEVEL LANGUAGE	LOW/MEDIUM	LOW	LOW
MANY HIGH LEVEL LANGUAGES	MEDIUM	MEDIUM	MEDIUM/HIGH
HIGH LEVEL LANGUAGE AND LOW LEVEL CODING (FALLBACK)	LOW	MEDIUM/HIGH	HIGH

FIGURE 5
LANGUAGE STANDARDS

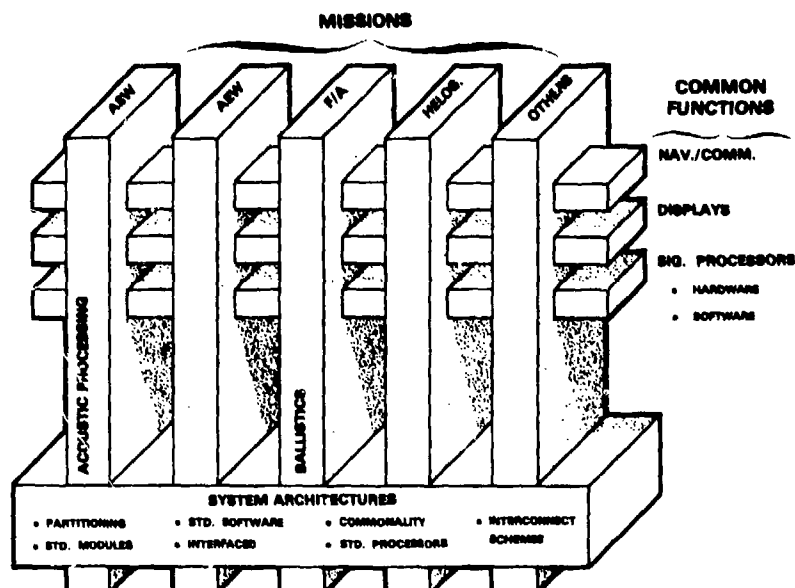


FIGURE 6
**STANDARDIZATION - ARCHITECTURE
INTERACTION MATRIX**

FUNCTIONAL DOCUMENTATION - A PRACTICAL AID TO THE ORDERLY
SOLUTION OF THE SYSTEM DESIGN PROBLEM

J.T. MARTIN

FERRANTI COMPUTER SYSTEMS LIMITED,
Western Road, Bracknell, Berkshire, England.

SUMMARY

This paper describes a method of breaking down a Customer Requirement in an orderly manner so as to produce progressively more detailed design levels such that at any one stage of the System Design the particular part of the design under consideration can firstly be easily understood and secondly comparatively isolated from the other parts of the design.

The most important characteristic of the design methodology is that the Requirement is considered in purely Functional terms until a highly detailed level of the design is reached. An example of this design methodology and the technique of Functional Documentation is given and the paper concludes by discussing the advantages that can accrue from a sensible use of the design methodology.

1. INTRODUCTION

To produce a successful design the system designer must start his design from the viewpoint of what the customer requires and work down to what sub-systems are required to achieve this requirement - the Top Down approach.

In order to carry out this Top Down design in a logical, structured, way it is important that:

- (a) the overall problem is decomposed in a controlled fashion
- (b) that each layer of the design is considered in the correct level of detail such that on the one hand sufficient information is considered before moving to the next lower level of design, while on the other hand the particular level of design reached is not unduly cluttered by consideration of too much detail.

The method used by Ferranti Computer Systems Limited to achieve these ends is Functional System Design utilizing as a tool in this process the powerful Functional Documentation (FD) technique.

The basic concepts behind design phase documentation were developed by the Systems Effectiveness Laboratory of Technical Operations Incorporated, Burlington, Mass. and further amplified by the United Kingdom Royal Navy.

The system of Functional Documentation, first used by Ferranti Computer Systems Limited (FCSL) as a tool for the design of extremely complex shipborne distributed processing systems, is now being used to carry out the demanding task of System Design for modern airborne distributed computing systems.

The main fundamentals of FD will now be described together with a brief example of the use of the technique.

2. THE PURPOSE OF DESIGN DOCUMENTATION

A fundamental tenet of the FCSL approach is that design and the documentation of that design are inseparable. The production of design documentation is, therefore, an integral part of the design task and the design documentation is itself a most important aid to the design process. At each stage of development, existing design documentation forms the basis for further development.

Until system development and implementation begins, the design documentation is the only tangible evidence of the intentions of the design team. It is a paper model of the system.

Furthermore, considering the project as a whole, design documentation must meet the needs of all subsequent stages through development, production, integration, installation and trials to post-delivery support. The design documentation must, therefore, provide not only a description of the proposed system, but also information necessary for the preparation of test requirements, trials specifications and servicing and maintenance data.

The successful design and development of an integrated system requires that each member of the design team be aware of the current design intent of his colleagues. In practice this implies a real time documentation system with a language common to the three principal disciplines involved, namely hardware, software and user.

Functional Documentation (FD) is this common language for use when the three disciplines must work in co-operation. It is a design tool developed specifically to assist in co-ordinating the design and through-life development of real-time systems. The formal documentation system of FD channels the paperwork output of system, hardware, software and user engineering staff into a standard format which is circulated amongst the understood and agreed by the design team. During the Project Definition phase it is the only available evidence of progress.

The completed FD forms an agreed document which defines the required system functions and their interfaces, and the inter-relationships between the disciplines. It is the specification of the system which shall be implemented by the individual disciplines during the Project Development phase. Hardware FD (H/D), Software FD (SFD) and User FD (UFD) are the languages employed at that stage when the individual disciplines may validly be developed independently.

The primary purpose of FD is, therefore to make and communicate the definitive statement of design. In fulfilling this it also achieves the following objectives:

- (a) To assist in the correct breakdown of the design into separable tasks and to logically define the scope, boundaries and interfaces of each task before the commencement of that task. As a result of performing each task, not only is technical progress achieved, but more detailed tasks are defined which also fit into the overall structure.
- (b) To allow technical analysis of the design. The documentation effectively provides a paper model of the system at all design phases and is constructed so as to highlight areas where design may be doubtful, inconsistent, ambiguous or incomplete. It is particularly suitable for examining the hardware/user/software interfaces and for predicting the reliability and maintainability of the proposed system. It further allows an individual designer to visualise the implications of design changes on related areas.
- (c) To enable technical management and the customer to monitor the progress of design, both to ensure that timescales and workloads are satisfactory and that technical requirements are being achieved. It is especially amenable to the use of PERT.
- (d) To provide a standard communications medium between the members of design teams working in different disciplines and in different companies.
- (e) To provide a permanent record of design, as it proceeds. It allows, for example, new staff joining the project to appreciate the philosophy, limitations and state of the design with a minimum of effort, and equally reduces disruption when members leave the project.
- (f) To provide a smooth transition of technical data into the maintenance handbook.
- (g) To form a basis and design record for subsequent Post Design (PDS) activities.
- (h) To provide an entry to HFD, UD and SFD.

3. FD PRINCIPLES

FD is a technique of logical and ordered technical description which uses graphical and pictorial presentation, supported by the written work, as the communications medium. It therefore takes advantage of the precision inherent in diagrammatic presentation.

To provide clarity of technical description, the subject matter is sub-divided in two dimensions, which are known as the "level" and the "function".

(a) Level. As the design phase of a project develops information becomes available in increasing degrees of detail and complexity. By its nature, early phase information is more general. It is said to be "high level" information and is by definition the first to be documented. Subsequent information may be classified as either "intermediate" or "low" level.

These levels are assigned a numerical reference as follows:-

Highest level	:	1
Intermediate levels	:	2
		3
		etc.
Lowest level	:	n

The number of levels actually required for a complete description will depend on the complexity of the subject, and the amount of original design/development work.

(b) Function. The function, in the general case, is defined as that grouping of hardware, software and user necessary for the achievement of a required event or events. This implies that any possible division of a subject into its hardware, user or software boundaries is of secondary consideration. Functions will exist at all levels but the "required event" will become increasingly detailed at lower levels. For example, "target destruction" may be a valid event at a high level and "status bit set" may be equally valid at a lower level. Therefore each function described at higher levels will be progressively sub-divided and amplified at lower levels.

3.1 Level/Function Relationship

Each function identifies a number of sub-functions which are then "expanded", i.e. described separately in greater detail, as functions at the next lower level. The process of sub-dividing the functions is repeated until a sufficient level of detail is reached. This is the lowest level, which will identify sub-functions performed solely by hardware, software or user.

The pyramid of functions formed by the progressive sub-division of the overall function is termed an hierarchical structure. Each function at each level is at once:-

- (a) A statement of requirement for its sub-functions at the level below, and
- (b) A definition of the implementation of the requirement stated at the level above.

Each piece of information necessary to define the design has a correct place in this logical structure, so missing information is highlighted, and there should be no duplication between levels or functions. Information is, therefore, rapidly accessed and easily retrieved.

3.2 FD Formats

At each level, and for each function at that level, information is produced in three categories which are mutually dependent. These are:-

(a) Functional Block Diagram (FBD). The function as defined by the previous level is expanded into its various sub-functions, and the inter-relationship of these sub-functions is defined in terms of sequence and information flow. The sub-functions are then each subsequently further developed on individual FBD's at the next level.

(b) Functional Text (FT). The FBD is supported by text, including a concise statement of the purpose of each sub-function, which may be presented in the corresponding physical location on the page facing the FBD, as Functional Blocked Text (FBT).

(c) Supplementary Information (SI). This category covers all other information not readily assimilated into the first two categories. It includes such data as physical layouts, channel allocations, manning requirements, design theory if applicable etc.

It is the FBD which makes the definitive statement concerning the function and its scope, whilst the FT or FBT is of a supporting nature. Any additional information may be presented as SI.

3.2.1 FD Document

The FT, FBD, FBT, and SI (in that order) for each function may be made up with a covering front sheet or comment sheet and master page/distribution list as an FD document. These individual FD documents then fit into the hierarchical structure and build up the complete Functional Documentation for the system.

3.2.2 Functional Reference

All functions at all levels (except Level 1, the top level) are assigned a functional reference of the form F1.2.3.4... The number of numerical digits in the reference defines the level at which the FD document for that function will appear.

Thus F1 appears at Level 2
F1.2 appears at Level 3
F1.2.3 appears at Level 4

etc.

Since every function at, say Level 2 is expanded at Level 3, so each functional reference is expanded also, retaining the digits of the original functional references.

If a given function requires more than one FBD document to expand it meaningfully at the next level, then the subject is divided up as befits the circumstances. The functional references for, say, three FBDs would appear as F1.2.3/1, F1.2.3/2, F1.2.3/3 and the overall reference at the previous level would appear as F1.2.3/1.3. This facility, which should not be abused, is however very useful for describing a function which has for example more than one mode (e.g. normal and reversionary modes) or more than one phase of operation.

Clearly it is advantageous if documents normally issued separately, e.g. trials schedules, interface specifications, requirements specifications, are published as further Supplementary Information. These documents are then indexed by the functional reference and are accessible from the hierarchical structure.

4. FD EXAMPLE

Annex 1 to this paper presents an example of the use of the FD technique.

The example shows how part of a system (Function 33 of the overall system in the example given) is first simply described at level 1 with only the main attributes of the function, as seen externally, described. The overall function is considered to be dividable into three (in this example) sub-functions which are then each considered in more detail at the next level down (Level 2). This process of functional division can be carried out until it is possible to define the process that each box on the FD is to perform in either hardware, software or user terms. In the example given it can be seen that each box on the level 2 diagrams can be used to produce the necessary specifications to enable the implementation of the particular box in question.

5. CONCLUSION

The Functional Documentation system allows the orderly decomposition of an overall system specification into progressively more detailed levels of design information. At any particular level of the design the amount of information to be handled is manageable by, and understandable to, the designer tasked with the job of furthering the design. The technique allows people from the different disciplines working on a

project (user, hardware, software) to communicate together in a commonly understood descriptive format. The technique forces a complete examination of the system specification and exposes any deficiencies or omissions that may exist. Functional Documentation allows the customer full insight into the design, shows him which areas of the original specification are insufficiently precise and also shows him which parts of the specification are most difficult to accomplish and perhaps candidates for re-examination in terms of cost/complexity/requirement tradeoffs.

Using Functional Documentation allows a complete functional description of the design to be produced, and changed if necessary, before the prototyping stage is started. Design changes can be made by simply altering lines on paper, not by expensive re-design of hardware or software modules. At the end of the Functional Design stage the information necessary for the production of the hardware, software and user specifications is available, consistent and achievable.

ANNEX 1EXAMPLE OF THE USE OF FUNCTIONAL DOCUMENTATION

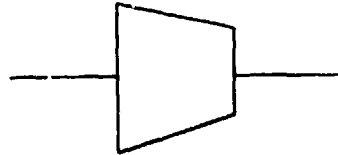
- 1.1 Functional Block Diagram Format
- 1.2 Example Functional Documentation Level 1
- 1.3 Example Functional Documentation Level 2 Function 1
- 1.4 Example Functional Documentation Level 2 Function 2
- 1.5 Example Functional Documentation Level 2 Function 3

ANNEX 1.1FUNCTIONAL BLOCK DIAGRAM FORMAT

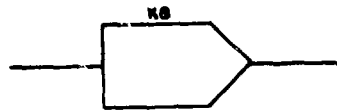
The diagrammatic format employed to describe the design is that of "Functional Documentation" as used within Ferranti Computer Systems Ltd. The following is a brief summary of the salient points:

(a) The diagram illustrate primarily the flow of information (be it data or control) between functions. Thicker lines are used to emphasise significant information paths. The diagrams are essentially time-sequential, left to right.

(b) Functions which are to be implemented by the "system", be it hardware or software (or as yet unknown) are illustrated by the symbol:



(c) Functions which are to be implemented by an operator action are illustrated as:



The device used in the operation is identified above the symbol, e.g. KB = Keyboard, TB = Tracker Ball.

The use of the operator function implies some interfacing hardware and software to get the information into the system. These hardware and software components (Service Functions) are omitted if they do not add to the understanding of the function in hand.

(d) Display of system information to the operator, is shown by the following symbol, annotated by the device type:



Again the Service Functions involved are omitted if non-critical to understanding of the application.

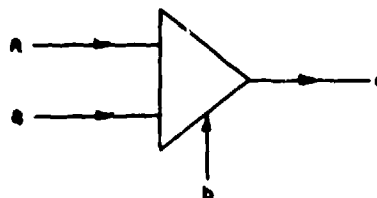
(e) hardware items are shown within dashed boundaries.

(f) Where other Application Functions are involved in the operation of the module, these are illustrated by the symbol:



The operation of these functions would be detailed elsewhere in the documentation.

(g) There is frequently a requirement to illustrate alternative paths for information flow. The following symbol is used:-



This illustrates either source A or source B being routed to destination C, subject to the control input D.

ANNEX 1.2EXAMPLE FUNCTIONAL DOCUMENTATION LEVEL 1MATCH CONTROL1. INTRODUCTION

MATCH (Medium-Range Anti-Submarine Torpedo-Carrying Helicopter) is a weapon system which utilizes a helicopter to carry and launch a torpedo in an anti-submarine engagement.

The control function is involved, primarily, in the calculation of the helicopter's course to fly and time to weapon release so that the aircraft controller can relay commands to the pilot. The procedure used for course and launch calculations is known as Vectored Attack (VECTAC). The calculations take account of the torpedo characteristics when deriving the aim point. The target position may be any track held by the system, or may be a fixed datum point indicated by tracker ball. Ship's radar is used to track the helicopter during its flight so that course corrections can be applied.

The function is also able to control a MAD Verification Run (MADVEC). In this application the helicopter is used to carry Magnetic Anomaly Detection (MAD) equipment to a suspected submarine position so that the presence can be verified or discounted by the change in magnetic field. This enables sonar contact which may in fact be shoals of fish, for example, to be eliminated. In a MADVEC the helicopter is guided to pass directly over the selected position, without any launch calculations.

Although the name MATCH indicates that a helicopter is used, it is also possible to employ a fixed wing aircraft without any differences to the operation of the function. Also it is possible to control an aircraft which is based on a consort rather than own ship.

Guidance of the helicopter is achieved by voice communication of the appropriate orders between the aircraft controller (ship's operations room) and the helicopter pilot. The pilot is responsible for launching the weapon by his own weapon controls.

Control of two MATCH engagements is possible at any one time. The two engagements must be controlled one by a North display operator and the other by a South display operator. A North (South) engagement may be taken over by another North (South) operator to allow for equipment failure. The Functional Specification does not describe two simultaneous engagements, as this simply implies two independent operations. A MATCH engagement can be controlled from any console which has facilities for communications with the aircraft.

2. MODE OF OPERATION

The function is sub-divided into three sequential phases of a MATCH engagement, see Block Diagram.

2.1 MATCH Preparation

This function is concerned with the insertion of certain parameters necessary for the calculation of a VECTAC and for the initiation of radar tracking on the helicopter. The actions involved can be carried out prior to and in anticipation of an engagement.

The following data is inserted:

- (a) Helicopter Indicated Air Speed (IAS).
- (b) Torpedo Initial Search Depth (ISD) - this is the depth from which the torpedo search becomes effective.
- (c) Torpedo Ballistic Correction - this indicates the distance the torpedo will fly between release and splash point.
- (d) Magnetic Variation - for helicopter course corrections.
- (e) Wind Data also for helicopter course corrections.

Tracking of the helicopter can either be carried out manually or by an auto-extractor. The normal close range surveillance radar, or the helicopter's transponder returns to the RRA equipment may be used, as appropriate to clutter and range conditions.

2.2 MATCH Approach

The approach function controls the engagement from the point of initiation until the final attack phase. Throughout this phase the function repeatedly re-calculates the course to be steered by the helicopter so that it will intercept the target allowing for increment of the target and drift of the helicopter. It is a basic assumption of the calculations that the helicopter will fly the adhered course from its current position at the pre-determined IAS and height, and that the target will maintain its last estimated heading and speed. The vector calculations performed do not therefore allow for turning circles of the aircraft nor for non-linear prediction of the target position and velocity.

The target may either be a track in the system or a fixed datum point inserted by an operator. The latter case allows for suspected target positions which are not being tracked, or for fleeting sonar contacts, etc.

The VECTAC calculation is performed every $4\frac{1}{2}$ seconds during the approach phase up to 13 seconds before weapon release where a countdown phase is entered (see Match Attack). It is essential for the operator to guide the aircraft on to a stabilised path during the approach phase, or else engagement will have to be aborted and a new Vectac initiated.

The Approach Function calculates the following data for display to the operator and transmission to the pilot:

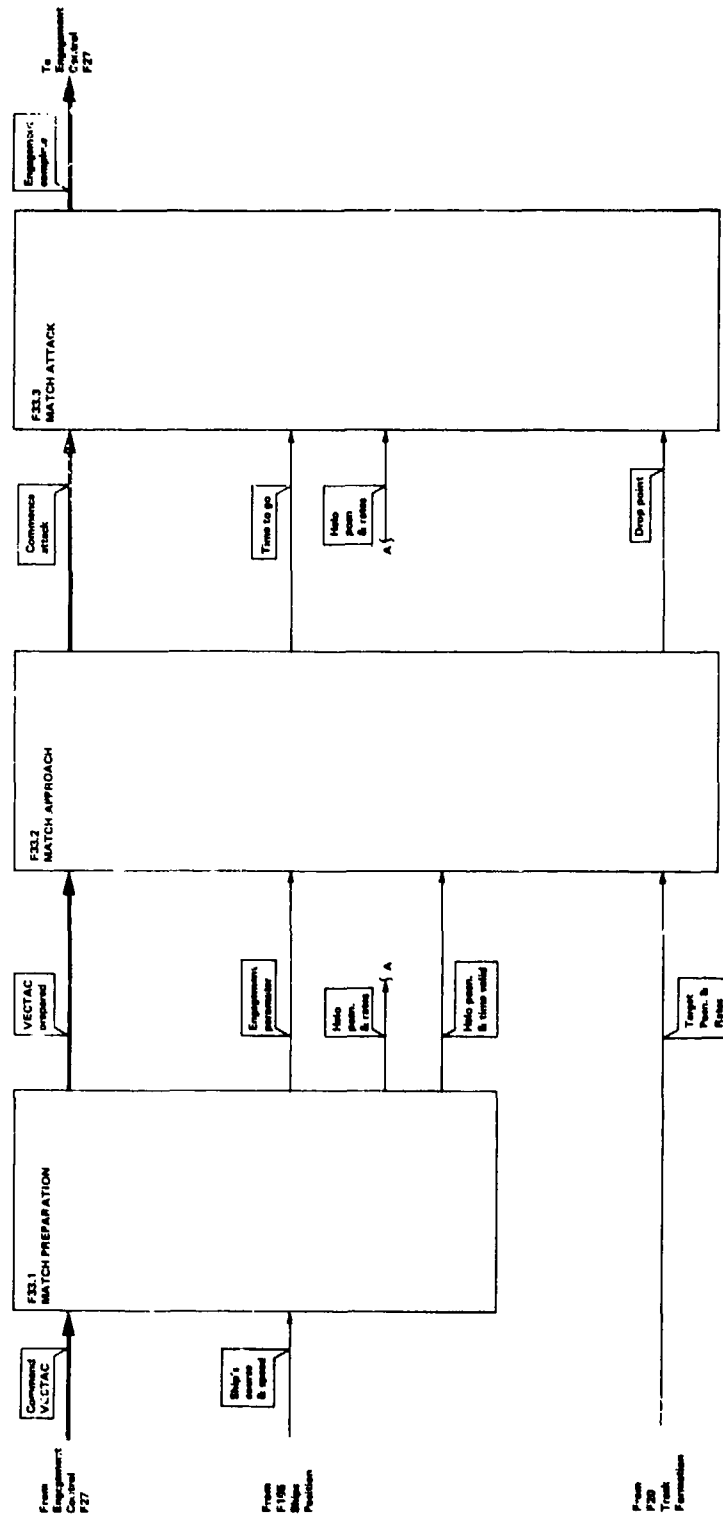
- (a) Course to Steer (CTS) Magnetic or True.
- (b) Distance to Fly to Weapon Release Point (DFG).
- (c) Time to go to Weapon Release Point (TTG).

It also controls the synthetic display of a Drop Point on the Controller's radar display.

2.3 MATCH Attack

The Attack Function controls the final approach of the helicopter on its current fixed course. During the attack, a countdown is relayed to the pilot so that he knows exactly when to release the weapon. The same procedure applies in a MADVEC since it is necessary for the pilot to know exactly when he is over the target area in order to record the MAD detection (or to mark the output of a pen recorder).

Once the weapon has been released the facility calculates where the torpedo will hit the water, and initiates the display of a splash point on the radar display, together with a surrounding weapon danger area (dogbox).



F33	MATCH CONTROL	ISSUE
LEVEL 1	FUNCTIONAL BLOCK DIAGRAM	DATE

ANNEX 1.3EXAMPLE FUNCTIONAL DOCUMENTATION LEVEL 2 FUNCTION 1MATCH PREPARATION1. INTRODUCTION

This function is concerned with the insertion of parameters necessary for the calculation of a VECTAC and for the initiation of radar tracking on the helicopter. The actions involved can be carried out prior to and in anticipation of an engagement.

2. MODE OF OPERATION (References refer to FBD)

Initiation of a MATCH engagement will generally come from the Anti-Submarine Warfare Director. On his command the variable parameters are manually injected into the system, led by the injection to convert (6)
Relative Wind to True Wind. The injections can be checked on the Check Line readout of the tote, or at any (2)
stage by query injections.

According to radar considerations, the operator will also select auto or manual tracking of the helicopter, (8)
thus implementing the Radar Manual Tracking or Radar Autotracking Functions. (11-13)

When all the variable parameters have been inserted, and the helicopter track has been initiated, the (14)
operator can proceed to the MATCH Approach phase.

3. DESCRIPTION

3.1 Conversion of Relative Wind to True Wind (1-4)

Relative wind is displayed on a VCS unit which indicates direction and speed (relative to ship's motion). The VECTAC requires true wind and hence the following injection is used to input relative and obtain true wind:

RW? W105P S18

"Display in the readout the True Wind direction and speed, where the relative direction and speed are as indicated (e.g. Red 105°, 18 knot)".

After the conversion, the readout is presented as:

W215 (direction to + 1°)
S25 (speed to ± 1 knot)

3.2 Insertion of True Wind (6)

The true wind velocity, required for calculation of helicopter's relative velocity, is inserted by the following injection, using data from the relative to true conversion:

HC + W215 S25

True wind value is as indicated (e.g. direction 215°, speed 25 knots).

3.3 Insertion of Ballistic Correction and Indicated Air Speed (6)

Ballistic Corrections are available to operators for each aircraft type which may be used in a Vectac. The correction is a distance value (horizontal displacement Weapon Release Point to Splash Point) which is applicable to the aircraft's Vectac engagement speed (IAS). The value applies to a preordained altitude at which the aircraft will fly.

The MI Control function requires these to be inserted as a parameter couplet in the form:-

HC + R16 S90

Ballistic Correction and Indicated Air Speed
Values are as indicated (e.g. 160 yards, 90 knots).

For a MADVEC, the Ballistic Correction is indicated as zero.

3.4 Insertion of Magnetic Variation (6)

The MATCH Approach Function automatically converts helicopter bearings to magnetic for relay to the pilot. The variation is injected as:

HC + V-7

"Magnetic Variation is as indicated
(e.g. 7° East)"

Note: If a True heading is required (as will be necessary for some types of helicopter) a variation of + or - should be injected.

3.5 Insertion of Initial Search Depth

(6)

Initial Search Depth (ISD) is required so that Vectac can allow for the time taken for the torpedo to reach an effective search position, when calculating the desired splash point. The value is injected as:

HC + U25

"ISD is as indicated (e.g. 250ft)".

For a MADVEC, the ISD value is indicated as zero.

3.6 Checking Vectac Parameters

(6)

Manual injections are also available to enable one of two groups of stored Vectac Parameters to be interrogated at any time:

HC? WSV

"Display in the readout stored values of True Wind Direction and Speed and the Magnetic Variation".

Readout (e.g. : W215
S035
V-07

(5)

HC? RSU

"Display in the readout stored values of Ballistic Correction, LAS and ISD".

Readout (e.g. : R016
S090
U250

(5)

3.7 Tracking the Helicopter

The operator selects the mode of tracking and the type of equipment to use according to conditions and whether the helicopter is on board or airborne. Reference should be made to Picture Compilation for detail on the tracking functions. If the helicopter is on deck and cannot be tracked normally, a manual track is set up alongside the ship for correlation with the helicopter when it is airborne. Alternatively, tracking may be initiated using the RRA equipment and the helicopter's transponder. The Vectac calculations can start as soon as initiation is made. It is possible to change the tracking mode from Manual to Primary to Secondary radar during any phase of a MATCH engagement without affecting the VECTAC.

(8,
11-
13)

ANNEX 1.4

EXAMPLE FUNCTIONAL DOCUMENTATION LEVEL 2 FUNCTION 2

MATCH APPROACH1. INTRODUCTION

The MATCH Approach Function calculates the VECTAC solution at regular intervals between initiation and the final attack phase. As a result of the calculations, the helicopter is guided to the target area.

2. MODE OF OPERATION (References refer to FBD)

The VECTAC calculation is commenced by the initiation command which may either specify a track number and a datum point for the target. (10)

The calculation is carried out once every $4\frac{1}{2}$ seconds until time to go (TTG) is less than 13 seconds. (11, 1-4)

As a result of the calculation, the aircraft controller's tote and labelled radar display are updated with engagement symbology and guidance commands for the helicopter pilot. (5-9)

The VECTAC can be cancelled prematurely if required.

3. DESCRIPTION

3.1 Selection of Target Type (10)

If the target is being tracked by the system, the VECTAC will be initiated using the target's and helicopter's track numbers. Otherwise the target position is marked by tracker ball.

3.2 VECTAC Initiation for Tracked Target (10)

Initiation of the VECTAC is made by the following Manual Injection:

HC? 4731 00465
"Calculate VECTAC for target and helicopter tracks indicated, and display the solution in the readout (e.g. target track 4731, helicopter track 0465)".

The target track may alternatiely be indicated by placing the tracker ball over the track and injecting:

HC? |TB| 00465

A North side VECTAC can only be initiated if a North side console is not already progressing a VECTAC (similarly for South side).

3.3 VECTAC Initiation for a Fixed Point Target (12)

The following manual injection is used to initiate a VECTAC on a fixed datum point:

HC? P|TB| 00465
"Calculate VECTAC for the fixed position using the indicated helicopter (e.g. 0465) and display the solution in readout".

Similar restrictions to the initiation apply for this operation, as are explained in Paragraph 3.2.

3.4 Cycling the VECTAC Calculation (11)

Every VECTAC Calculation provides a solution with which the helicopter can be controlled. In order to allow for changes in course of the target, and deviations from defined course by the helicopter, the calculation is repeated every $4\frac{1}{2}$ seconds during the approach phase.

3.5 The VECTAC Calculation (2-4)

Refer to Figure .

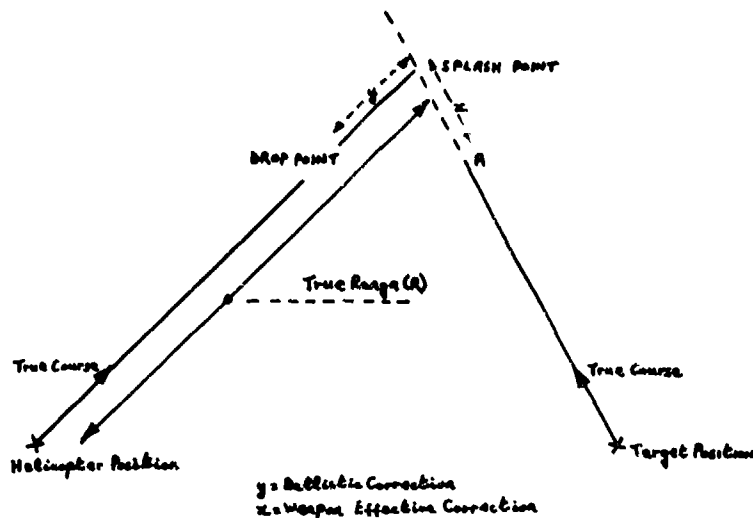
The aim of the VECTAC calculation is to make the helicopter reach the Splash Point (SP) at the same time as the submarine reaches point A. The distance x (Weapon Effective Correction) is such that the torpedo will fall to Initial Search Depth in the same time as the submarine takes to reach the Splash Point (it is assumed that the torpedo travels vertically downwards once it is in the water).

In order for the torpedo to enter the water at the splash point, it is necessary for the weapon to be released at the Drop Point (DP). The distance y, known as the Ballistic Correction is supplied as a VECTAC Parameter and is dependent on the aircraft and weapon type. Ballistic Correction is not corrected for wind during the calculation.

Once the distance x is known and the current position of the submarine and helicopter have been derived, the vector velocity solution can be calculated. Other factors available or derived for the solution are:

Helicopter IAS
True Wind
Target True Course and Speed

The result of the vector calculation is the indicated course of the helicopter. There are generally two solutions to this calculation (a quadratic), the function adopts the solution which provides the maximum closing relative velocity.



3.6 Calculation of Splash Point

(2)

The time taken for the torpedo to reach ISD is calculated from the following algorithm:

$$t = 53 + \frac{\text{ISD}}{53} \text{ seconds}$$

where ISD is in feet.

With t , the distance x is calculated, knowing the target speed.

If an ISD of Zero is injected the value of t is set to 10 seconds.

3.7 Calculation of Helicopter Course

(6)

The vector solution is applied to calculate the helicopter indicated course, employing the following data:

Helicopter current position
Target current position
Target true velocity
Helicopter IAS
True Wind velocity
Weapon Effective Correction (x)

If there is no solution to the calculation the function causes the indication NOGO to be displayed for the current calculation cycle.

(6)

3.8 Calculation of Helicopter Control Data

(4)

Once the vector solution has been obtained the following values are calculated:

True Range (R)
True Distance to Drop Point ($R-y$)
Time to Drop Point (TTG)
Indicated Distance to Fly (DTG)

The Ballistic correction value defines y .

If TTG is calculated to be greater than 9 minutes 59 seconds the VECTAC is considered unfeasible and NOGO is displayed.

(6)
(1)

3.9 Calculation of Magnetic Course

The magnetic course value is required by the pilot, and is calculated from the Helicopter course.

3.10 Controller's Tote Readout

(5,6)

During a MATCH Approach a readout of the following form is produced:

0465 (Helicopter Track Number)
C105 (Helicopter Course, e.g. 105°)
0075 (Distance to Fly, e.g. 7.5 dms)
5400 (Time to Weapon Release, e.g. 5 min. 0 sec.)

3.11 Controller's LRD Display

(7,8)

The Helicopter and target tracks will be on display, as supplied by the Track Formation function.

The MATCH Approach Function supplements the display with a synthetic marker for the Drop Point, in the form of an asterisk.

3.12 Guidance of the Helicopter

(9)

The controller relays the data on his readout to the pilot to enable him to steer the helicopter.

3.13 Detection of Completion of MATCH Approach

(4)

When TTG reaches 13 seconds the Approach phase is complete, so the calculation cycle is terminated and control passes to the Attack Function.

3.14 Premature Cancellation

The VECTAC can be cancelled prematurely by use of the standard, "cancel readout" injection, e.g. "DR-". This effectively clears the inhibit on further VECTACS described in para. 3.2.

ANNEX 1.5EXAMPLE FUNCTIONAL DOCUMENTATION LEVEL 2 FUNCTION 3MATCH ATTACK1. INTRODUCTION

The MATCH Attack Function controls the final phase of a MATCH engagement when the helicopter is on a steady course and counting down to Weapon Release Time. It also provides information to the controller when the torpedo has been dropped, for tactical evaluation.

2. MODE OF OPERATION

Refer to the Function Block Diagram.

The Attack phase is initiated by the signal Commence Attack which is received 13 seconds before Weapon Release. This signal starts the countdown in seconds. (1)

The countdown display on the tote is relayed to the pilot by the controller. When the countdown reaches zero, the pilot launches the weapon. (2 -4)

Once the weapon has been released the Drop Point display is removed and a Splash Point and Weapon Danger Zone (Dogbox) are painted instead. This enables the controller to assess whether the engagement was accurate, with reference to the target track. (5 -9)

3. DESCRIPTION

3.1 Countdown Control

The MATCH Attack Function is controlled by a one second countdown which initiates a tote update until TTG equals zero. (1)

3.2 Countdown Readout

(2,3)

During the attack the tote readout only shows the countdown value:-

0007 (1st line, e.g. 7 seconds)

This value is relayed to the pilot by the aircraft controller. (4)

3.3 Action at Weapon Release Time

When TTG = 0 the display of the Drop Point is removed and instead a splash point and dogbox are painted. The Helicopter pilot operates the launch controls for the torpedo. The inhibit on further VECTACS is removed at this time. (5-10)

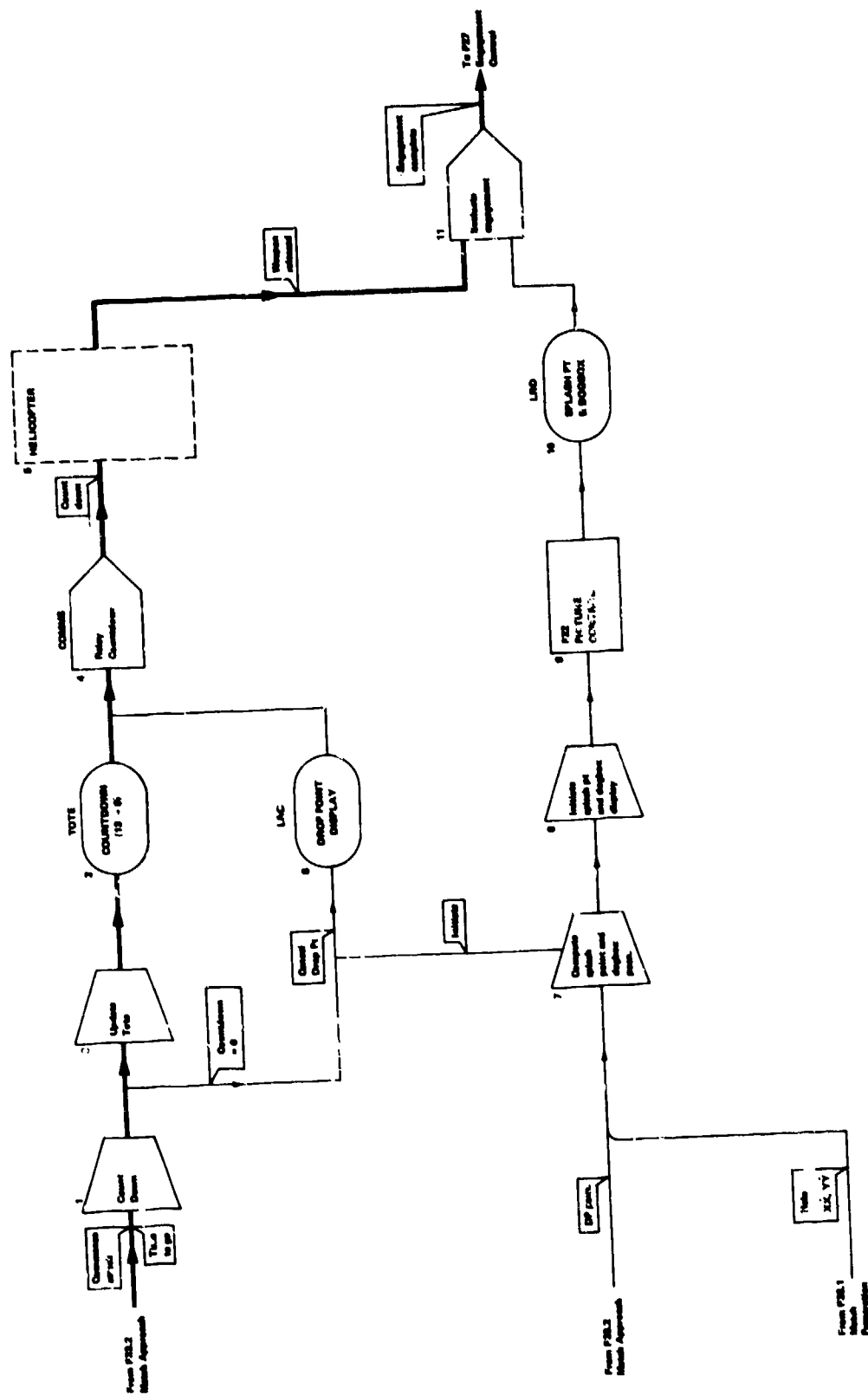
3.4 Splash Point Display

The splash point co-ordinates are calculated and a request is made for the Picture Control Function to generate a Splash Point and Dogbox. These two markers are deleted automatically after 7½ minutes by Picture Control. Alternatively they may be cleared on demand by the appropriate Picture Control Injection. (7-10)

3.5 Terminating the Engagement

The aircraft controller evaluates the engagement according to his display data and the helicopter pilot's report. A new engagement may be initiated by the ASWD if desired. (11)

If no new engagement is required the operator should clear the 0000 readout by injecting "DR-".



F21.3	MATCH ATTACK	ISSUE
LEVEL 2	FUNCTIONAL BLOCK DIAGRAM	DATE

DISCUSSIONS
SESSION III

REFERENCE NO. OF PAPER: III-11

DISCUSSOR'S NAME: J. T. Martin, Ferranti

AUTHOR'S NAME: A. A. Callaway

COMMENT: I notice that the program uses a constant overhead to allow for command and status words and the response times of the terminals. Have you considered allowing the response time to be a variable rather than a constant, the variable being specified on a per RT basis, if required? This would allow known response times of terminals to be inserted. On the same basis, are you intending to extend the program so as to cope with acyclic messages? In such a case, I would recommend that the program accept as a parameter the amount of time allocated to all acyclic messages and produce as an output the average and maximum waiting time before the acyclic message is handled.

AUTHOR'S REPLY: I feel that the average value of overhead is sufficient for the purpose, especially since the controller/terminal and terminal/terminal overheads are resettable parameters. The inter-word and inter-message gap figures preset into the program are 6 μ s - thought to be a reasonable average value between 4 μ s and 10 μ s which MIL-STD-1553B specifies. If the user observes that his terminals involve a spread of response times, then a representative average value will still suffice in the analysis - if the system is critically affected by this, then it probably needs some redesign anyway.

With regard to the second point, it is intended to extend SAVANT into the acyclic regime, and Mr. Martin's suggestion is gratefully noted.

REFERENCE NO. OF PAPER: III-11

DISCUSSOR'S NAME: Jim McCuen, Hughes Aircraft

AUTHOR'S NAME: Tony Callaway

COMMENT: Can SAVANT be modified and expanded to model a contention protocol, high speed data bus operating at 50 megabits?

AUTHOR'S REPLY: SAVANT includes resettable parameters - for example, the transmission bit rate, word overheads, message overheads, word length, and message length. These can be changed to any values characteristic of the protocol one wishes to investigate. For example, we have used it at RAE to estimate traffic on a 50-megabit time slot bus protocol.

REFERENCE NO. OF PAPER: III-12

DISCUSSOR'S NAME: K. Brammer, ESG

AUTHOR'S NAME: H. J. Whitehouse

COMMENT: Are you aware of any applications of systolic array processing to nonlinear optimal recursive filtering?

AUTHOR'S REPLY: Systolic arrays can be used for the computation and inversion of the covariance matrices associated with Kalman filtering. In the area of nonrecursive nonlinear filtering, systolic arrays whose elements can perform comparisons can be used for rank-order filtering, especially median filtering.

REFERENCE NO. OF PAPER: III-14

DISCUSSOR'S NAME: H. P. Kahlen, G. E.

AUTHOR'S NAME: J. T. Martin

COMMENT: To be compatible with your "design documentation," I would like to know how an "ideal" specification should look? More functional diagrams or the "old-fashioned" item-by-item specification?

AUTHOR'S REPLY: It does not really matter in what format the requirement specification is presented. The important thing is to ensure that the requirement specification specified fully those things that you require. The specification should not request items that are not required but tend to be put into the specification because the customer has a preconceived idea of what the design should look like. For instance, if you require a computer to be able to store data in a non-volatile store then state that in the requirement, do not specify that a core store should be used--that is not the requirement in this case. Of course, if you have a particular reason for needing to use a defined piece of equipment, for instance so as to provide compatibility with some other unit, then this is one of your mandatory requirements and should be included in the specification.

REFERENCE NO. OF PAPER: III-14

DISCUSSOR'S NAME: DR. N. J. B. Young, Ultra Electronic Controls

AUTHOR'S NAME: J. T. Martin

COMMENT: You have talked about an aid to system design, but not covered testing or post-development modifications to the customer's specification. It is our experience (in Ultra Electronic Controls Ltd.) that these absorb a very high percentage of costs. Can you tell us something of how your system design aid can be applied to system (hardware and software) testing and post-development modifications, and whether it makes them easier or more difficult?

AUTHOR'S REPLY: Functional documentation helps you to move from the original system requirement specification to the specification necessary for the hardware and software required to produce the system. Test specifications for the system come from the FD because the FD describes in an easily understandable form the attributes that must be proven to exist.

If the customer's specification changes after development, the functional documentation lower levels can be used to discover whether the change to the specification will affect the software or the hardware or both and also help to choose between a change to the software or a change to the hardware if a choice exists.

As stated above, FD is used in the overall system design phase, those attributes which the system must exhibit are defined, data structures, processing functions, and crew actions to meet this defined system requirement are detailed on the FD, if required, to make the system function in the required manner, otherwise they are left to the hardware, software, or user design stages.

Functional documentation does produce a good interface between the designer and the customer, because it is so understandable. However, it also allows the design to proceed in a logical top down manner and thus offers all the other advantages listed in the paper.

REFERENCE NO. OF PAPER: III-14

DISCUSSOR'S NAME: J. P. Quemard, Electronique Marcel Dassault

AUTHOR'S NAME: J. T. Martin

COMMENT: Trois remarques sur la méthode présentée

- pas d'aspect structure des données
- pas de regroupement fonctionnel des traitements
- pas de gestion de references croisées, de chromogrammes

N'est ce pas simplement une façon de présenter une documentation pour le client plutôt qu'une méthode de travail.

Three remarks on the method presented:

- not mentioning the data structure aspect
- not mentioning the processing functional regrouping
- not mentioning the cross-reference management, chromograms

Isn't it merely a method to present a documentation to the customer/user rather than a working method?

AUTHOR'S REPLY: The response to this question is included in the response to Dr. Young's question.

REFERENCE NO. OF PAPER: III-14

DISCUSSOR'S NAME: Dr. van Keuk, AVP Member

AUTHOR'S NAME: J. T. Martin

COMMENT: I would agree with you that in the early phase of system design you may not need any computer assistance. The main reason may be that many of the ideas to be invoked are unsharp. But, of course, in the software design phase computer assistance in large systems is necessary to define the data organization, of positioning, and things like these.

AUTHOR'S REPLY: The question asked was whether the use of interactive computer display techniques would be useful for producing the functional documentation designs. The answer is - not at all. When the FD diagrams are produced, each diagram is the combined effort of a number of people working together using pencils on a piece of paper. The initial diagram produced is very rough, not even rulers are used to draw the lines. When the diagram has been produced by the engineer it is redrawn by a technical author but this is a small task.

The paper concerned itself with only system design. Functional documentation is used at the system design stage. Other techniques are used for software, hardware, and user designs. I would just briefly state that computer-aided design techniques and simulation are useful in all the three areas.

A CONSISTENT APPROACH TO THE DEVELOPMENT OF SYSTEM REQUIREMENTS AND SOFTWARE DESIGN

By

A. O. Ward
British Aerospace Public Limited Company
Aircraft Group
Warton Division
Preston
PR4 1AX
United Kingdom

SUMMARY

Some of the problems encountered in the development of system and software requirements are discussed and generalised solutions suggested. A specific approach is described, the SAFRA Project, including extensions into the area of software design. This approach embraces the use of a new methodology, Controlled Requirements Expression (CORE) interfaced with a computer based System Description Language for storage and automatic analysis. Software design assumes the use of a MASCOT rationalised executive and CORAL as the implementation language. Experimental procedures for the automatic extraction of CORAL programmes from detailed requirements held on a Database are discussed.

The techniques are illustrated via an example based on the processing associated with a Fuel Management System.

1. INTRODUCTION

1.1 Problem Areas

During the latter half of the previous decade there was a growing awareness of the importance of the requirements for embedded software, particularly for large projects with lifecycles extending over many years. Two of the major problems are the desire for realisation leading to insufficient resources being allocated to the requirements phase and the apparent inability to communicate the requirement effectively to the implementer.

A more specific case for examining the way in which requirements are developed may be made by noting that the cost of changes to software increase over the lifecycle particularly when many of the errors which precipitate such changes may be traced to inadequacies in requirements and design. Also, traditionally a relatively small percentage of the procurement budget is devoted to requirements and so a large absolute increase in the resources devoted to requirements will lead to a relatively small percentage increase in the overall cost of a project. These three pieces of evidence suggest that increased investment in the early stages of projects involving embedded software will potentially have a large cost leverage on their success. Unfortunately this is difficult to achieve because although funding can usually be found to solve critical problems just prior to entering service it is hard to convince people of the worth of investment early on in projects.

There are a number of areas in which requirements can be improved, some outside the scope of this paper, those addressed by the technique described here are discussed below.

A heavy reliance on the use of a natural language invariably leads to ambiguity. English, although semantically a very rich language, is notoriously open to interpretation. Imposing a detailed format on a requirement document goes some way towards alleviating this problem but if the detail itself is communicated using English the problem will still remain.

Projects of reasonable size will inevitably lead to the requirements phase being undertaken as a team activity and this in turn causes problems when attempts are made to assess the consistency of the various documents produced. Again, the use of English and the lack of any detailed structure prevent the use of formal methods to check for consistency. The many stages involved in current implementations not only make it difficult to demonstrate conformance but increase the risk of errors due to the many communication interfaces that have to be crossed. Finally, requirements are usually incomplete and as suggested earlier this is usually due to insufficient effort rather than a lack of methodology or formalism. However, there are some classes of information which, due to the conventional form of requirements, it is almost impossible to check for completeness. In addition, current approaches do not have the mechanism for accommodating viewpoints which although seemingly irrelevant at the early stages of a project will become very relevant once the system approaches service.

1.2 General Solutions

These problem areas we believe can be addressed by the following means. Ambiguity can only be solved by using a precise method of expression, such as the diagrammatic notation shown in Fig. (1). Here, the simple expedient of representing processes in boxes, data on arrows and depicting time ordering as left to right across the paper provides an unambiguous picture of the relationships between the processes and their sequence.

Validation for consistency is impossible without a visible information structure. If the simple system shown in Fig. (1) is to be described in such a way as to assist a consistency check then a suggested information structure could be as follows:

PROCESS: Centre of Gravity Calculation;
 PART OF: Mass C of G Calibration;
 USES: Individual Tank Fuel Mass;
 Wing Sweep Position;
 DERIVES: Fuel Centre of Gravity;
 COMES AFTER: Mass Calculation;

The reserved words in capitals are a selection of specific information categories to which have been assigned the objects and relationship shown in the diagram. Clearly the checking of two descriptions for consistency is simplified by having such a structure and this is illustrated in Fig. (2) where two inconsistent descriptions of the functions depicted in Fig. (1) are given and some of the errors highlighted. Because of its mechanistic nature this process is amenable to automation provided we have access to a language which can be used to describe the structure and a database in which to hold the information.

Conformance can only be preserved and demonstrated if the requirement has a structure which allows this. Such a structure, of course, will correspond to the various stages of development as well as the detailed steps through each stage and the documentation produced. A good analogy is a notional aircraft drawing scheme. Here a general arrangement (GA) will be originated from which, at the next level, some features will be represented in a little more detail via a feature G.A. The latter will be broken down into assemblies and in turn to sub-assemblies, the final stage being a detailed part which can be handed to the implementer for manufacture. One can observe that there are several levels of detail and at each level the customer and designer assess in turn whether the design is practicable, will satisfy the requirement and if it is correct. The hierarchy of information that each drawing level represents can be seen to be a logical decomposition of the preceding levels and there is an unambiguous method of expressing the design (i.e. a drawing system with standards). In addition the interrelationships between various levels and drawings at the same level are referenced on the diagrams.

When applied to the development of system and software requirements it is clear that such an approach, if applied rigorously, would enable conformance to be established via a series of small increments of detail. Equally, the effect of changes to the requirement can be quickly traced through the hierarchy in order to establish the functions affected by such a change.

Finally, completeness is satisfied mainly by the allocation of adequate resources and sufficient time to the requirements phase, however, as stated earlier mechanisms for partitioning work with complete interfaces in a team activity must be sought.

A three element solution to the derivation of requirements which will alleviate these problems is provided by the use of a:

- . Methodology
- . Standard
- . Automated Aid

and we will discuss these briefly in turn.

The methodology should encompass the process by which requirements are both developed and expressed. It must be usable by engineers, as opposed to systems analysis in the traditional sense, and have a notation which is not only simple to use but is relatively transparent to the technical content. The latter is important from the customers point of view where it should not be necessary for him to have a detailed understanding of the methodology in order to undertake technical reviews of the documentation produced. It should be applicable to any stage of system conception as far as the customer will allow. It should impose no constraints on design decisions but rather provide the necessary cues to the engineer so that such decisions are made at the appropriate level of detail at the correct time.

The standard should provide the information structure and the tests to be made against such a structure. These quality control actions should be matched by rigorous configuration control procedures. Automated aids to the application of the standard via the use of computer based tools to implement the mechanistic aspects of the quality control procedures should have a minimal impact on the prime method of expression used by the engineer. Where the tests cannot be carried out automatically, reports should be provided which give maximum assistance to the engineer in checking his requirement. One should also aim to minimise the data preparation task. The important elements of such a tool are shown in Fig. (3).

The notation used for expression is described using a System Description Language (SDL) via the information structure specified in the standard. Once the requirement is in this form it may be checked using an Input Analyser which not only assesses the validity of the input in its own right but also its consistency with information already held on the database.

Once held on the database it may be subjected to the repertoire of reports available for analysis or interfaced with other tools.

1.3 Software Design Interface

The software design interface is yet another barrier to successful communication of needs and we should seek to minimise discontinuities by aiming for a more gradual transition between requirements and implementation. If possible the notation and methodology used in the requirements phase should be consistent with those used during design. Also, if use is made of a rationalised (and ideally standard) executive to specify software module communication and control it should be possible to produce a more formal route map between requirements and basic design.

Similarly if a standard Higher Order Language (HOL) is employed then this argument can also be used to justify a similar formalism between detailed requirements and the subsequent software.

In the next section we will attempt to describe a specific approach to requirements and design which it is hoped goes some way towards satisfying the above. Some aspects of the approach are still experimental and these will be highlighted in the discussion.

2. SPECIFIC APPROACH

2.1 SAFRA Project

A specific approach to requirements and software design is suggested by the SAFRA project. Semi Automated Functional Requirements Analysis (SAFRA) is a proposed approach to requirements and software design, consisting of existing methods and tools and new ones currently under development. A more detailed picture of the background to SAFRA and its initial objectives and assumptions can be found in Ref. (1). If we consider a phased life cycle approach (Fig. (4)) then what is proposed is a consistent set of methods and tools for each phase. These will be described below in as much detail as this paper will allow but with reference to the discussion above they may be summarised as follows.

The methodology used by the engineer to develop and express the requirement is Controlled Requirements Expression (CORE). This is a new technique developed jointly by B.Ae. Warton Division, and Systems Designers Ltd., embracing a method for the assembly and analysis of information relevant to a requirement and an easily understood diagrammatic notation as the method of expression.

The automated aid to validation and storage of both requirement and software design is the University of Michigan's Problem Statement Language and Problem Statement Analyser (PSL/PSA).

The software design interface assumes the continuing use of CORE notation to produce detailed specifications with storage using PSL/PSA but aimed at the use of a rationalised executive and HOL. The former is the Modular Approach to Software Construction Operation and Test (MASCOT) and the latter is the UK MoD standard CORAL 66. A further assumption is the use of a commercially available MASCOT based software development and test environment for the testing phase working on the host-target principle.

2.2 Controlled Requirements Expression

CORE is a method of analysing and expressing requirements in a controlled and precise manner. It enables a subject requirement to be expressed as either a number of lower level requirements or as a component part of some higher level. Any lower level requirement derived using CORE may in turn be subjected to the method to produce a hierarchy of lower levels. The lowest level is that at which the full method need no longer be applied and one may resort to strictly hierarchical decomposition making use of the notation alone. This is considered to occur after the basic design stage has taken place. In general the same notation is employed at all levels of requirement and design and some of the major symbols are illustrated in Fig. (5).

CORE diagrams utilise boxes to represent processes and arrows to represent data. The diagrams are time ordered from left to right and thus the box ordering specifies the sequence in which processes must occur. Symbol free boxes shown in parallel indicate indeterminate order and overlapping indicates a number of identical processes occurring in parallel. All input data entering a CORE diagram is referenced to a source and all output data to a destination.

Data arrows may also be used to describe repetition, selection and condition. Those arrows appearing from the top of a process box point to a reference which indicates that this process is functionally equivalent to the one described at the reference. Those appearing at the bottom of a process box indicate the mechanism that performs the process. Iteration is shown by an asterisk in the top right-hand corner of a process box and mutual exclusion by a small circle in the top left-hand side.

The method comprises eleven logical steps which when applied to a subject requirement will decompose it into its lower level components and these are summarised below.

The method has three stages for each level of decomposition which may be summarised as

- Information Gathering
- Propose Relationships
- Prove Relationships

Information is gathered with respect to a number of subdivisions of the problem, referred to as Viewpoints, in terms of input and output data and gross functions. This information is refined by a Data Decomposition Step which specifies in more detail the data already tabulated.

Relationships are proposed between the inputs and outputs for each Viewpoint in turn and for data flowing across the Viewpoint, and these are termed 'Single threads'. The proof of such relationships is done in two ways; first the interrelationship between Viewpoints is examined and where such links exist new diagrams in the form of 'combined threads' are drawn. Secondly, as threads represent only particular paths through system operation and in no way depict such aspects as parallelism or the operational time ordering of processes another diagram is required which will illustrate this. This is achieved by the construction of a 'combined operational' diagram or examining how threads interact operationally across Viewpoints.

Both of these will lead to iterations through the previous steps precipitating a more detailed examination of the single threads for correct combination and in order to establish operational relationships.

The outcome of the above is a partitioned description (in terms of Viewpoints) with a detailed and hopefully complete picture of how the Viewpoints interrelate and react with each other as well as some indication as to the major functions contained within them. It is now possible to extract the Subject Viewpoint as the one of interest and in turn take Viewpoints which describe how it is composed and repeat the methodology in full.

Such decompositions continue until functions emerge which may be seen to be implemented as software on a particular processor, once decisions have been made regarding the computing elements. At this stage it is possible to enter into basic design, but before discussing this phase we must say a little about MASCOT.

2.3 MASCOT

The definition of MASCOT given in Ref. (2) describes it as a set of facilities for real time programming incorporating features concerned with systems development and construction, achieving this by providing the following:

- (a) A formalism for expressing the software structure of a multi programmed or real time system which can be independent of computer configuration and programming language.
- (b) A disciplined approach for design, implementation and testing which provides a concept of modularity for real time systems and added reliability brought about by increased control over access to data.
- (c) An interface to support the implementation and testing methodologies which is provided by a small kernel that can either be implemented directly on a bare machine (for operational use) or on top of a host operating system (useful for system prototypes) as well as software construction facilities.
- (d) A strategy for documentation.

MASCOT, as Ref. (2) continues, views an application system as a number of activities, or processes, which operate independently and asynchronously, but which cooperate by accessing shared Intercommunication Data Areas (IDAs). Thus the system can be seen as a network whose nodes are the activities and the IDAs whose directed links are pathways for data flow between activities and IDAs.

Although the MASCOT facilities allow great variety in the implementation of IDAs, it has been found useful, for design purposes, to distinguish two conceptual classes of IDA according to the nature of the data flow which they support. These are called channels and pools. A channel supports unidirectional data transmission, it has an input interface associated with a number of producer activities and an output interface associated with a number of consumer activities. A pool provides a permanent data area in which data remains available for activities to read until it is specifically overwritten. The data in a pool does not have the essential transience of channel data and reading it does not imply consumption, conceptually it has a simple bi-directional interface with associated activities.

MASCOT system designs are represented by Activity-Channel-Pool (ACP) diagrams and Fig. (6) shows the symbology along with a simple example. Clearly, the logical outcome of a basic design phase using MASCOT would be a set of ACP diagrams showing the identified activities and how they are related through appropriate IDAs. Integrating a requirements phase using CORE with a basic design phase using MASCOT specifically means changing a requirement (a CORE) diagram into a design (an ACP) diagram. This area of methodology is still in the very early stages of development but it is possible to postulate two possible approaches to this step.

- A software designer takes the CORE diagram as a statement of the requirement and by considering the constraints of processor throughput, memory available etc., produces what he views as an optimum basic design in the form of an ACP diagram. The design, while reflecting the software architecture, will retain the functional relationships specified in the requirement and because of the structural method of expression in use it will be possible to demonstrate that the basic design conforms to the needs of the requirement.
- The second and perhaps controversial approach is to draw parallels between data relationships in the CORE sense and those between activities in MASCOT. In short, propose a direct correspondence between a CORE diagram and an ACP diagram and thus minimise the software design step in the traditional sense. One might suggest that this approach is only feasible where performance constraints do not exist.

However, let us assume that by either means the design diagram has been produced, subsequent steps consist of further decomposition of each activity or software function making use of CORE notation. At each 'layer' of this detailed description the diagrams are encoded in PSL, checked and stored on the database.

The terminating layer is one where the diagrams reflect logic which is directly transcribable to a programming language, in this case CORAL, however the transition is done automatically by use of a specially designed suite of PSA reports and a formatter.

2.4 PSL/PSA

This topic has been left until now so as not to break the continuity of the discussion on the transition from requirements to design. PSL/PSA is a System Description Language, analyser and database developed as part of the ISDOS project at the University of Michigan. The reader is referred to Ref. (3) for a discussion of its background and a description of the more important features. In the context of SAFRA, PSL/PSA is being employed in two specific ways.

Conventions have been established for encoding particular subsets of CORE data sets (i.e. Tabular Entries, Combined Threads etc.) into PSL and running suites of reports to check their correctness. Such passes are part of the quality control procedures demanded by the standard. The second area relates to the detailed specification of software functions at the activity level and below, in order to produce a database of the specification which may then be used to automatically generate the programmes which satisfy the root procedure that corresponds to the activity. PSL consists of a large number of reserved words pertinent to particular aspects of system description, these are summarised below and examples of those relevant to processes in particular are given in Section 3.4.

- Communication and analysis
- System boundary and input/output flow
- System structure
- Data structure
- Data derivation and manipulation

- System size and volume
- System control and dynamics
- Project management

To complement the language there is a suite of 32 reports available, each one related to the aspects given above. The reports fall into four categories, Indented Lists, Matrix, Structure/Chain and Function Flow and some simple examples are shown in Fig. (7).

3. ILLUSTRATION VIA AN EXAMPLE

3.1 Introduction

The examples given below are the result of two separate studies addressing requirements and design as separate entities but for convenience they are presented here as the result of a contiguous series of phases. This obtains because they are the result of two separate development phases for the methodology dealing initially with requirements and then addressing the interface with a software design phase and the production of programmes.

The example shown here is a Fuel Management System (FMS), or specifically the processing associated with an FMS, and we will now say a few words about this requirement.

The customer input was the hardware system layout diagram shown in Fig. (8) and the need was to generate the requirement for an associated control system. The design aim was to produce an automatic FMS to reduce pilot workload. It should have the capability of initiating the normal transfer sequence but should also be able to recognise faults and reconfigure the transfer sequence accordingly.

The agreed assumptions for the system's starting point were therefore:

- Twin engine aircraft
- Six fuel tanks - Forward Fuselage
 - Rear Fuselage
 - Left Wing
 - Right Wing
 - Left External
 - Right External
- Transfer; all tanks shall be capable of transferring fuel to forward and rear tanks.
- Asymmetry; fuel asymmetry shall be automatically controlled to provide constant asymmetry between forward and rear tanks.
- Information should be provided to enable the ground crew to service the aircraft via a Ground Service panel.
- Single failure shall not be catastrophic.
- Sufficient information shall be made available to the crew to enable interaction if and when required.
- System shall perform automatically and provide self diagnosis and self correction capability.

3.2 Development of Requirements

The requirement was developed through two levels of decomposition. Level 3 and its associated layers transcend the traditional interface of software design although in this approach it is seen as a continuing decomposition of the requirement in order to produce a language independent description. However, for convenience level 3 will be discussed separately. A schematic representation of the hierarchy is shown in Fig. (9). The objective was, by use of the methodology to first establish the interaction between the complete FMS and other systems and then at a subsequent level establish the interaction between the processing element of the FMS and other parts of the subsystem. The third level, then, corresponds to a detailed description of the functions satisfied by the FMS processing element.

The Viewpoints selected at level 1 were as follows:

- Allied Command; the actions external to the aircraft of providing fuel, mission and co-ordination data etc.

- Other Aircraft Systems; those which have an influence on or an area influenced by the fuel system (i.e. cooling systems, engine systems etc.).
- Environment in which the aircraft operates.
- Fuel Management System, embracing pumps, valves, pipes, tanks, processors, embedded software etc.

The Viewpoints selected at level 2 were as follows:

- Data Highway, the method of transferring the data from sensors to the processor and from the processor to controls and displays etc.
- Controls and Displays, the pilot and ground service panels.
- Fuel Management System Process, all management and control functions to be embodied in software.
- Fuel Handling, hardware such as pumps, valve, pipes, tanks etc.

Some brief examples of the CORE methodology will now be discussed. Fig. (10) shows two aspects of information gathering associated with the second level of decomposition. A Tabular Entry for the Viewpoint of FMS Processing is shown with a decomposition of the data passing between the FMS Processing and Fuel Handling Viewpoints. The additional level of detail in the latter allows the former to be examined and a number of actions associated with the data identified. The proposed threads, an example of which is shown in Fig. (11), are identified for all the data listed. This particular thread, In Flight Refuel Control Actions, is given for the Viewpoint FMS Processing and the data links identified interface with the other three viewpoints as well as that data identified at the previous level as flowing across the problem boundary. The interaction between this proposed thread in the Fuel Processing Viewpoint and that of other threads in other viewpoints is now examined by considering these data relationships. When particular or thread relationships can be found a Combined Thread diagram may be constructed as shown in Fig. (11). Here, two threads proposed in the FMS Processing Viewpoint are shown interacting with In Flight Refuel Valves Actions in the Fuel Handling Viewpoint, data passing into and out of the diagram demonstrate the relationships with other threads.

These diagrams have been drastically simplified from the original documents in order to help communicate a feeling for the methodology, they also represent a very small sample from a three volume data set. The final step is the construction of the Operational diagram (Fig. (13)).

3.3 Basic Design

As stated in 2.3 the basic design phase consists of producing a design diagram from the CORE requirement diagram either as a specific software design activity (i.e. with recourse to optimisation) or by establishing a direct correspondence between the two diagrams via their data relationships. For interest we will discuss the latter approach, while bearing in mind that it may produce a less than optimal solution, and consider the CORE Operational diagram shown in Fig. (13).

There are similarities between some of the data relationships used in CORE and those assumed in the use of MASCOT. Specifically we believe there is a correspondence between what are termed Thread and Associated Thread relationship in the CORE domain and channels and pools in the MASCOT sense. The Operational diagram may be redrawn with the appropriate notation for channels and pools once the configuration of specific pools has been decided. Such a conversion is shown in Fig. (12) and corresponds to a CORE Design diagram. In principle this differs from a MASCOT ACP diagram by having activities in rectangles rather than circles.

3.4 Detailed Design

The consequence of the previous steps are a number of software functions, represented by threads, with an operational view of how these threads interact in terms of a requirement. This requirement has been used to establish a Basic Design consisting of Activities and their associated IDAs. For simplicity we will now consider one of these Activities, specifically MASS-CALCS and show the steps undertaken to carry out a detailed definition of the sequential process that supports this Activity. In MASCOT terminology this is called a Root Specification and the first description of the MASS-CALC Root Spec derived at level 2 is shown in Fig. (14).

Further decomposition of the processes shown on this diagram will not only provide the appropriate macro expansion but also give a more detailed breakdown of the data structures associated with the channels and pools. Such decomposition is done strictly hierarchically and the only resort to CORE is the use of the diagrammatic notation. Such a decomposition is a lengthy business and the subsequent tree transcends six layers. The terminating box on each branch of the tree corresponds

to an expansion of the macro referenced in the box. Layers above this termination are expressed as CORE diagrams and the information they contain is encoded in PSL for both data and process. The nature of the information stored in this way is reflected in the example shown below, via the process CALC IND TANK FUEL MASS part of the MASS CALCS ACTIVE composition of Fig. (14). Note that words on the left-hand side of listing correspond to PSL reserved words.

```

DEFINE PROCESS                                CALC-IND-TANK-FUEL-MASS;
/* DATE OF LAST CHANGE - JAN 26, 1981, 11:29:25 */
SYNONYMS ARE: B1P06;
ATTRIBUTES ARE:
    REPEAT-RANGE SET-OF-TANKS,
    ORDER          3,
    TYPE            'REPEATED-MACRO-CALL';
SUBPARTS ARE:  CALC-ONE-TANK-FUEL-MASS,
                PUT-ONE-TANK-MASS,
                ADD-TO-RUNNING-TOTAL,
                ONE-PROBE-PRELIM-MASS-CALCS,
                LOOKUP-ATT-CRCTN-FACTORS,
                WRITE-FUSE-TANK-FUEL-MASS;
PART OF:        MASS-CALCS-ACTIVE;
CREATES:
    CALC-IND-TANK-FUEL-MASS-LOCAL;
DERIVES:        IND-TANK-FUEL-MASS;
DERIVES:        USABLE-FUEL-MASS;
USING:          DENSITY-CRCTN-FACTOR;
EMPLOYS:        BUS-ATTITUDE-DATA,
UPDATES:        TANK-ID;
USING:          TANK-ID;
UPDATES:        TANK-RUNNING-TOTAL;
USING:          TANK-ID,
                TANK-RUNNING-TOTAL;
INCEPTION-CAUSES:
    LOOKUP-ATT-CRCTN-FACTORS;
TERMINATION CAUSES:
    PUT-TOTAL-FUEL-MASS;
ON TERMINATION OF:
    SET-TANK-RUNNING-TOTAL-ZERO;
PROCEDURE;
C 'FOR' TANK ID := LWT 'STEP' 1 'UNTIL' RFT 'DO'
N      CALC IND MASSES (TANK, ID, DCF, GAUGING DATA, RUNNING TOT,
N      ATTITUDE DATA,
N      IND MASSES, USABLE MASS)-
E;

```

The above may be seen to consist of six areas:

- The process name corresponds to that found on the diagrams with an appropriate synonym, in this case a keying index which allows the process to be traced back to the root of this particular tree, MASS-CALCS.
- Qualities of the process in the form of Attributes may be entered and the examples given here include the TYPE of process, a Repeated Macro Call with the value of the REPEAT-RANGE given as SET-OF-TANKS.
- Upward and downward hierarchical relationships, via the SUBPARTS and PART OF terms show that the macro has several constituent processes that will be defined at the next layer.
- Data relationships and their specific significance are represented by:
 - USING: Conventional utilisation of data local to the diagram (eg TANK-ID).
 - EMPLOYS: Conventional utilisation of data entering the diagram (eg GAUGING DATA).
 - DERIVES: Production of data which subsequently will leave the diagram. (e.g. USABLE FUEL MASS).
 - UPDATES: Iterated variable (e.g. TANK-RUNNING-TOTAL).
- Sequencing and control relationships, such as the first process to be triggered within the process CALC-IND-TANK-FUEL-MASS, signified by INCEPTION-CAUSES, here LOOKUP-ATT-CRCTN-FACTORS. Similarly the process to be triggered when this sequence of processes is complete, signified by TERMINATION CAUSES, here PUT-TOTAL-FUEL-MASS. Finally the process whose termination will start this particular sequence, signified by ON TERMINATION OF, here SET-TANK-RUNNING-TOTAL-ZERO.
- The last area, classified as PROCEDURE, will ultimately be reserved for mathematical expression which cannot be described using PSL. However, the currently experimental status of the PSA report suite and formatter being employed means that some aspects other than mathematical statements must be

included in CORAL at this time. The PROCEDURE statement is a comment entry on the database and hence is not amenable to being checked by the analyser.

3.5 Program Generation

Considering the MASCOT structure there are four types of program required to complete a system and these comprise:

- Root Specs, the actual program body for each activity and which form the bulk of the system.
- IDA Specs, (Pools and Channels) which include the body of Access Procedures.
- Module Declarations, which list all the components of the above giving the actual names to be used in the Form lists.
- Form Lists describe how the system goes together ie which pools and channels connect which activities to form (via subsystems) the complete system.

In principle all the information pertinent to these programs should be found on or derived from the database but here we will describe the steps concerned with a Root Spec and for simplicity only for the first macro.

The information needed to construct the first macro can be found in the PSL statements ATTRIBUTES ARE, PROCEDURE, TRUE-WHILE and FALSE-WHILE. An attribute, when applied to a process, will have names TYPE and ORDER, and when applied to data items will have names TYPE and CORAL-NAME. A process will correspond to a macro name and the attribute TYPE will thus describe the type of macro, in this case it is ROOT-SPEC (i.e. ROOT-SPEC being the VALUE of TYPE). A data item can be an ENTITY, GROUP or ELEMENT where the latter corresponds to an indivisible CORAL variable while ENTITY and GROUP are types of ELEMENT collections. Here the attribute name TYPE describes the CORAL variable for the purpose of making declarations within the first macro.

The structure of a macro is made up of Heading, Declarations, Calls and Close and the respective code elements are found as below.

Heading; in the PROCEDURE comment entry of the process with the macro name, the attribute name TYPE will have the value ROOT-SPEC.

- Declarations; in the attribute description of the appropriate GROUPS and ELEMENTS.
- Calls; in the PROCEDURE comment entries of the SUBPARTS of the first macro and the comment entries of the CONDITION section.
- Close; not in the database and thus created.

The approach adopted is:

- (i) Identify the macro name
- (ii) Trace the local variables from the macro name
- (iii) Trace the first layer subparts from the macro name.

Although this is the route for extracting the required information the order which the call code elements must take in the macro cannot be guaranteed. This has been overcome by the use of an additional attribute ORDER which provides the appropriate key.

The programme is generated by applying the above strategy through a suite of PSA reports, where the results of one report act as the file input to the next. The last step before submission to the compiler is a Formatter which deletes extraneous PSA messages accrued during the previous steps. An example of CORAL generated in this way is shown below representing the macro CALC-IND-TANK FUEL-MASS and calls to two layers below.

```

152 'COMMENT'      ==>MACRO          PSLNAME=CALC-IND-TANK-FUEL-MASS      ;
153 'DEFINE'
154     CALC IND MASSES (TANK ID,DCF,GAUGING DATA,RUNNING TOT,
155     ATTITUDE DATA,
156     IND MASSES,USABLE MASS)
157 "'BEGIN'
158 'COMMENT'      ==>MACRO          PSLNAME=CALC-IND-TANK-FUEL-MASS      ;
159 'FLOATING' 'ARRAY' ATT MASS E1:3?;
160 'INTEGER' PROBE;
161 'INTEGER' ONE TANK MASS;
162 'INTEGER' PROBE MASS;
```

```

163 'FLOATING' ATT FAC;
164     LOOK UP ATT FAC (BUS ATTITUDE DATA,TANK ID,
165                     ATT FAC);
166 'FOR' PROBE:= 1 'STEP' 1 'UNTIL' 3 'DO'
167     ONE PROBE PMC (PROBE,TANK ID, GAUGING DATA,
168                     ATT MASS);
169     CALC ONE TANK M (DCF,ATT MASS,
170                     ONE TANK MASS);
171     WRITE USABLE MASS(ONE TANK MASS,TANK ID,
172                     USABLE MASS);
173     PUT ONE MASS (ONE TANK MASS,TANK ID,
174                     IND MASSES);
175     ADD TO RUN TOT (ONE TANK MASS,RUNNING TOT);
176 'END';
177 'COMMENT'      ==>MACRO          PSLNAME=LOOKUP-ATT-CRCTN-FACTORS      ;
178 'DEFINE'
179     LOOK UP ATT FAC (BUS ATTITUDE DATA,TANK ID,
180                     AT FAC)
181 'BEGIN'
182 'COMMENT'      ==>MACRO          PSLNAME=LOOKUP-ATT-CRCTN-FACTORS      ;
183 'FLOATING' P,R,A,B,PITCH ANGLE,ROLL ANGLE;
184 'IF' TANK ID = LWT 'THEN'
185     'BEGIN'
186         GET ATT 'OF' DH IN(BUS ATTITUDE DATA);
187         PITCH ANGLE:=BUS ATTITUDE DATAEO7;
188         ROLL ANGLE:=BUS ATTITUDE DATAE17;
189         P:=(1/COS(PITCH ANGLE)-1);
190         R:=(1/COS(ROLL ANGLE)-1);
191     'END';
192 'IF' TANK ID = LWT 'OR' TANK ID = RWT 'THEN'
193     'BEGIN'
194         'IF' PITCH ANGLE 'GE' 0 'THEN' A:=P 'ELSE' A:=-P;
195         'IF' ROLL ANGLE 'LE' 20 'AND' ROLL ANGLE 'GE'-20 'THEN' B:=0      ;
196         ATT FAC:=1+(A+B);
197     'END'
198 'ELSE'
199     'BEGIN'
200         'IF' PITCH ANGLE<=30 'AND' PITCH ANGLE>=-30 'THEN' A:=0;
201         'IF' ROLL ANGLE<=20 'AND' ROLL ANGLE>=-20 'THEN' B:=R;
202         ATT FAC:=1+(A+B);
203     'END';
204 'END';

```

4. STATUS

As stated in the introduction the techniques described above are still in the development stage and the status of particular aspects are given below.

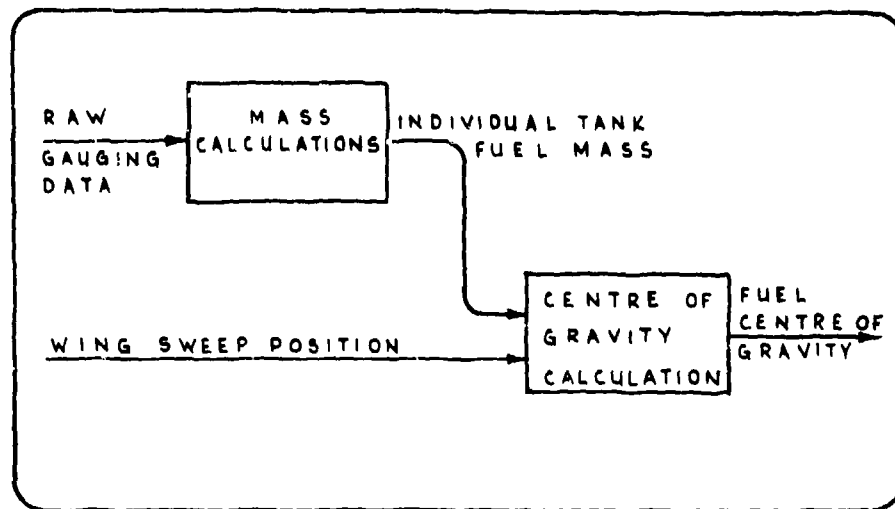
- PSL/PSA, MASCOT and CORAL are all commercially available and mature. PSL/PSA has been used extensively in the United States for the statement of requirements and MASCOT and CORAL have been employed on a number of real time projects in the United Kingdom.
- CORE as a requirements methodology, is being used on a number of small projects within BAe and its use continues to grow. Considerable effort has been expended in solving the transfer problem and an intensive training course is available to members of new projects.
- Experience to date has highlighted the problem of data preparation of PSL from CORE documentation as well as the control of the large data sets produced by the method. In order to solve both of these problems a computer based CORE work station is currently under development which will enable requirements to be developed at a terminal and automatically produce the associated PSL.
- The links with MASCOT and CORAL are experimental and a project currently being undertaken will seek to evaluate the conventions given above, as well as providing the means to produce a more powerful CORAL generator.
- Long term plans include interfacing CORAL with Ada including the automatic generation of Ada programmes.

5. REFERENCES

1. An Approach to the Derivation and Validation of Requirements.
A. O. WARD AGARDograph No. 258 Guidance and Control Software. May 1980.
2. MASCOT a Structured Software Design Methodology for Real Time Systems.
Infotech Seminar Nov. 1978.
3. PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems.
D. Teichrow and E. A. Hershey III. IEEE Transactions on Software Engineering. Jan. 1977.

FIG. 1 A SIMPLE DIAGRAMMATIC NOTATION

MASS C OF G DERIVATION



PROCESS: MASS CALCULATIONS

PART OF:

MASS CENT. OF GRAV
DERIVATION;

USES:

WING SWEEP POSITION

DERIVES:

INDIVIDUAL TANK FUEL
MASS;

COMES AFTER:

CENTRE OF GRAVITY
CALCULATION;

PROCESS: CENTRE OF GRAVITY
CALCULATION;

PART OF: MASS C OF G DERIVATION;

USES:

INDIVIDUAL TANK FUEL
MASS

DERIVES:

FUEL C OF G

COMES BEFORE:

MASS CALCULATIONS;

FIG. 2 INFORMATION STRUCTURE HIGHLIGHTS INCONSISTENCIES

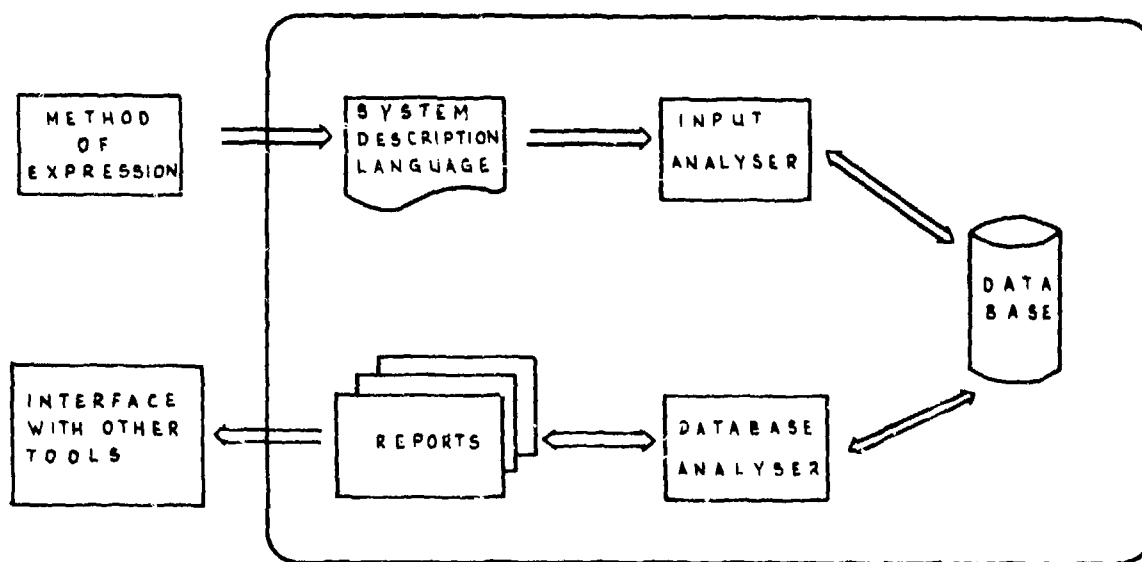


FIG. 3 MAJOR ELEMENTS OF A COMPUTER BASED TOOL.

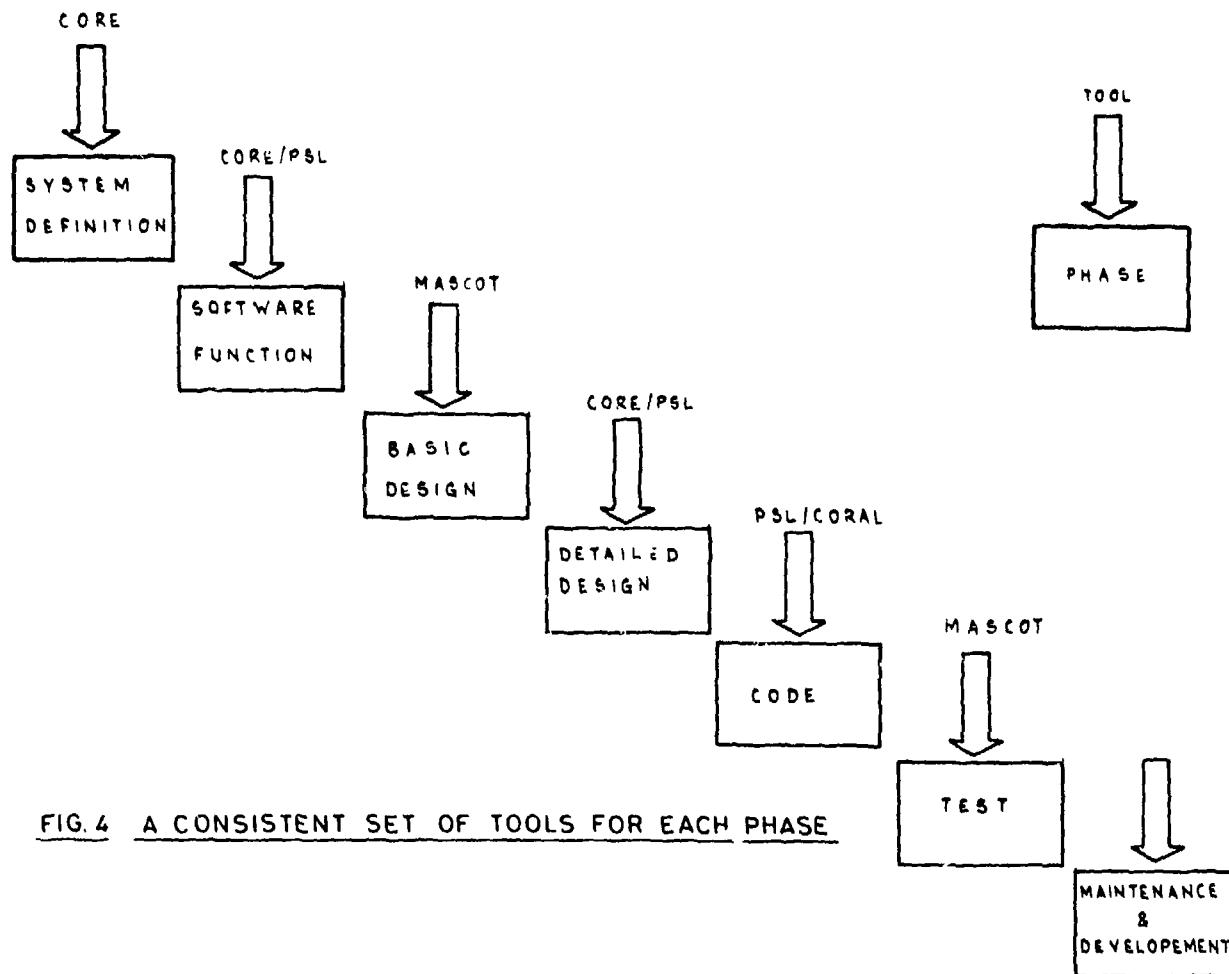


FIG. 4 A CONSISTENT SET OF TOOLS FOR EACH PHASE

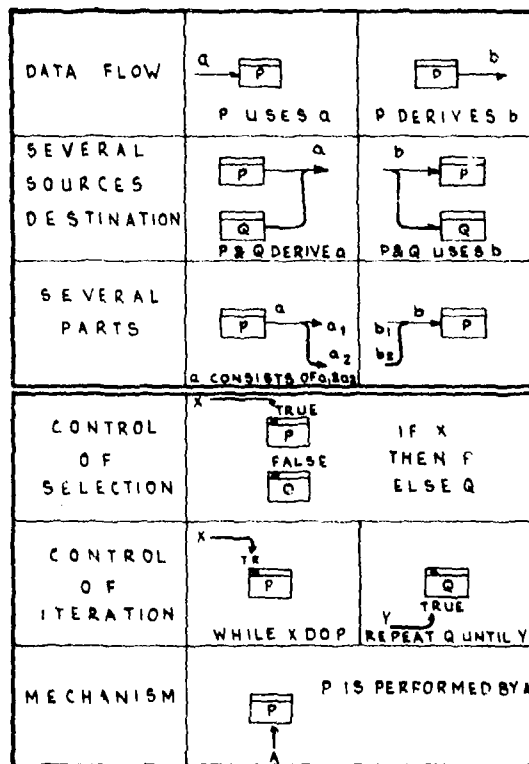


FIG. 5 EXAMPLES OF CORE NOTATION

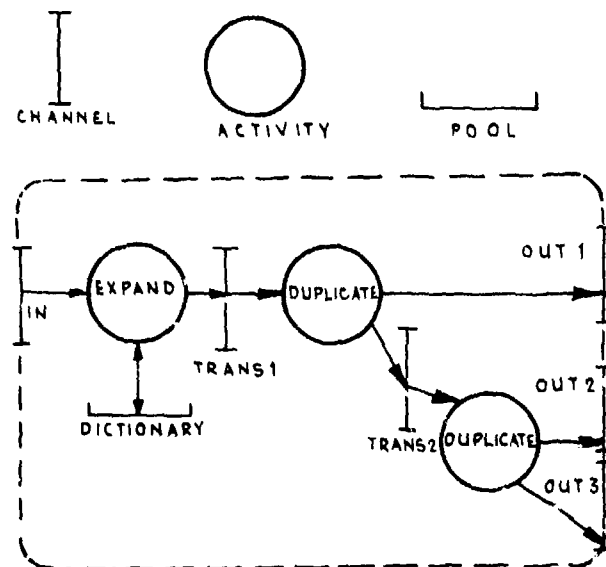


FIG. 6 EXAMPLE OF MASCOT NOTATION

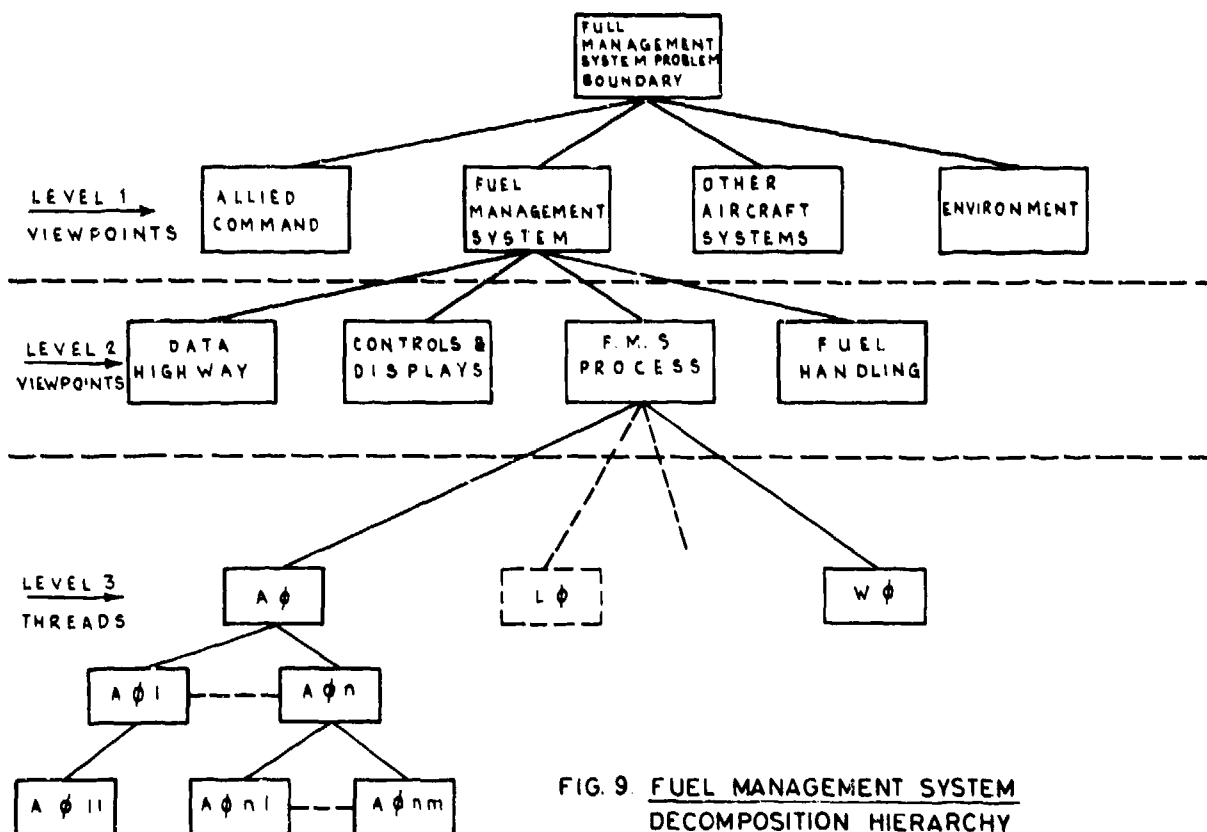


FIG. 9 FUEL MANAGEMENT SYSTEM
DECOMPOSITION HIERARCHY

SOURCES	OUTPUTS	ACTIONS	INPUTS	DESTINATIONS
DATA HIGHWAY	GND. PANEL VALVE STATE DATA FLIGHT TIME LEFT FUEL FLOW RATE WARNINGS FUEL ASYMM TANKS TRANSFERRING INDICATIONS FUEL DATA LEAK DATA VALVES STATE DATA FUEL C OF G ENGINE FEED PRESSURE	F. M. S. PROCESSING	BUS REFUEL DEMANDS BUS GND TRANSFER DEMANDS BUS DEFUEL DEMANDS BUS FUEL C OF G BUS A/C STATE DATA BUS INFLIGHT FUEL DEMANDS BUS DUMP SELECT	DATA HIGHWAY
FUEL HANDLING	CONTROL DATA		RAW VALVES POSN. DATA RAW SENSOR DATA	FUEL HANDLING

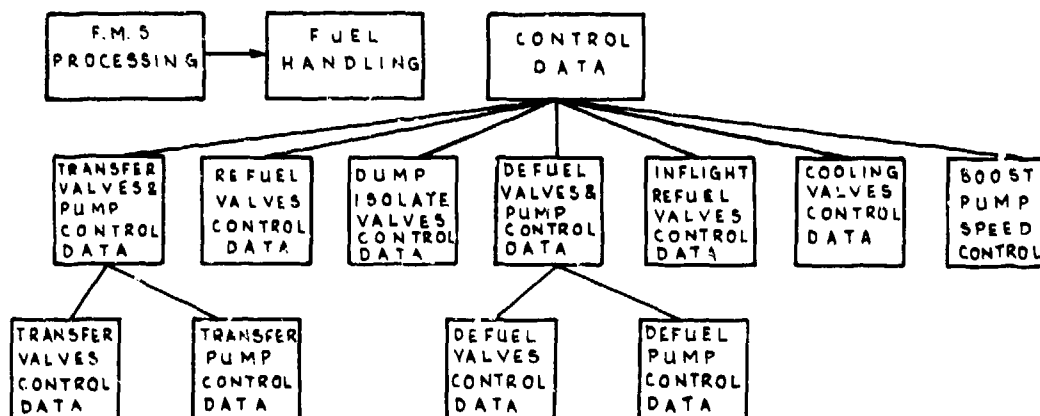


FIG. 10 MECHANISMS FOR GATHERING INFORMATION: TABULAR ENTRIES AND DATA

DECOMPOSITION.

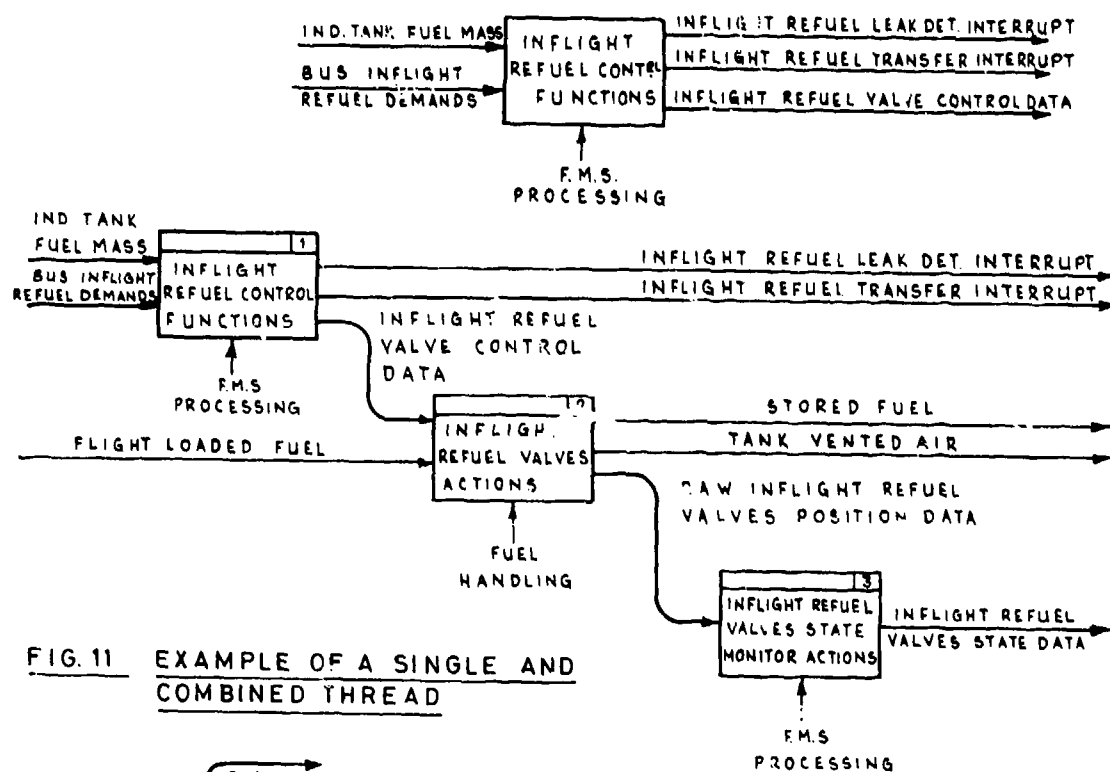


FIG. 11 EXAMPLE OF A SINGLE AND COMBINED THREAD

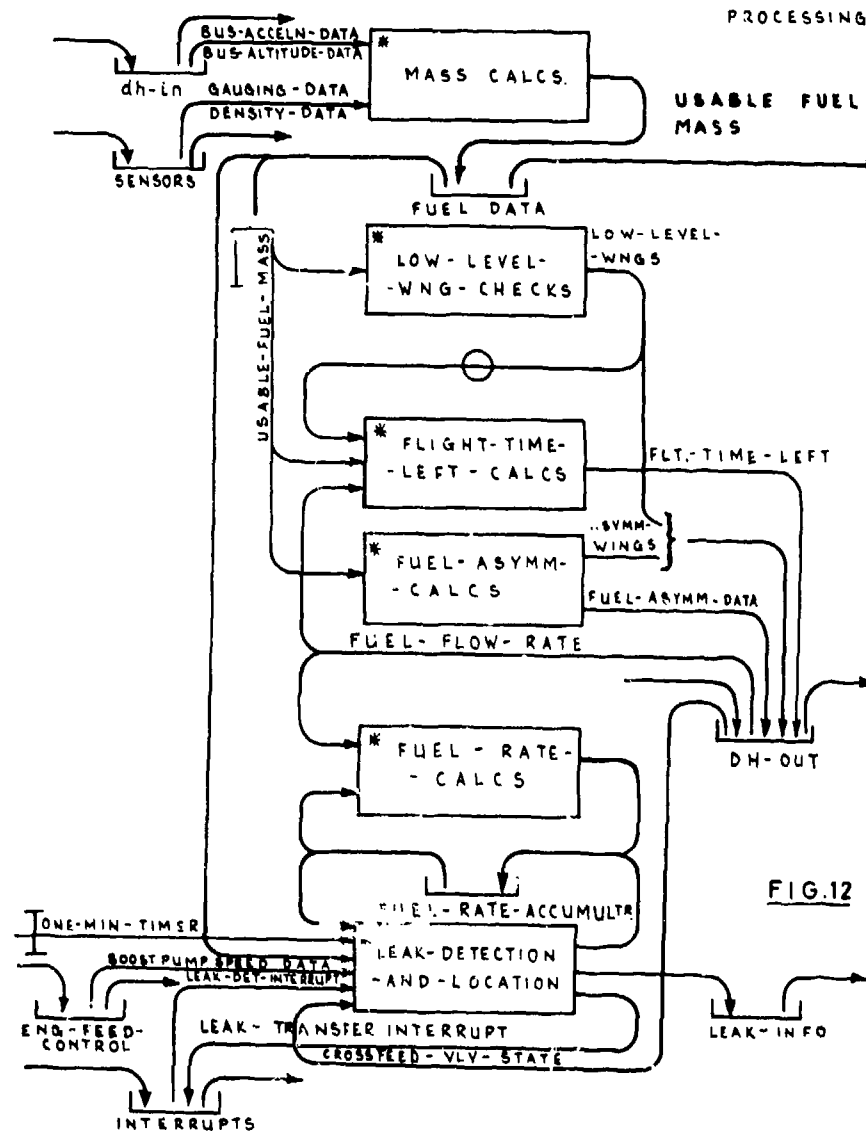


FIG. 12 SIMPLIFIED FMS DESIGN DIAGRAM

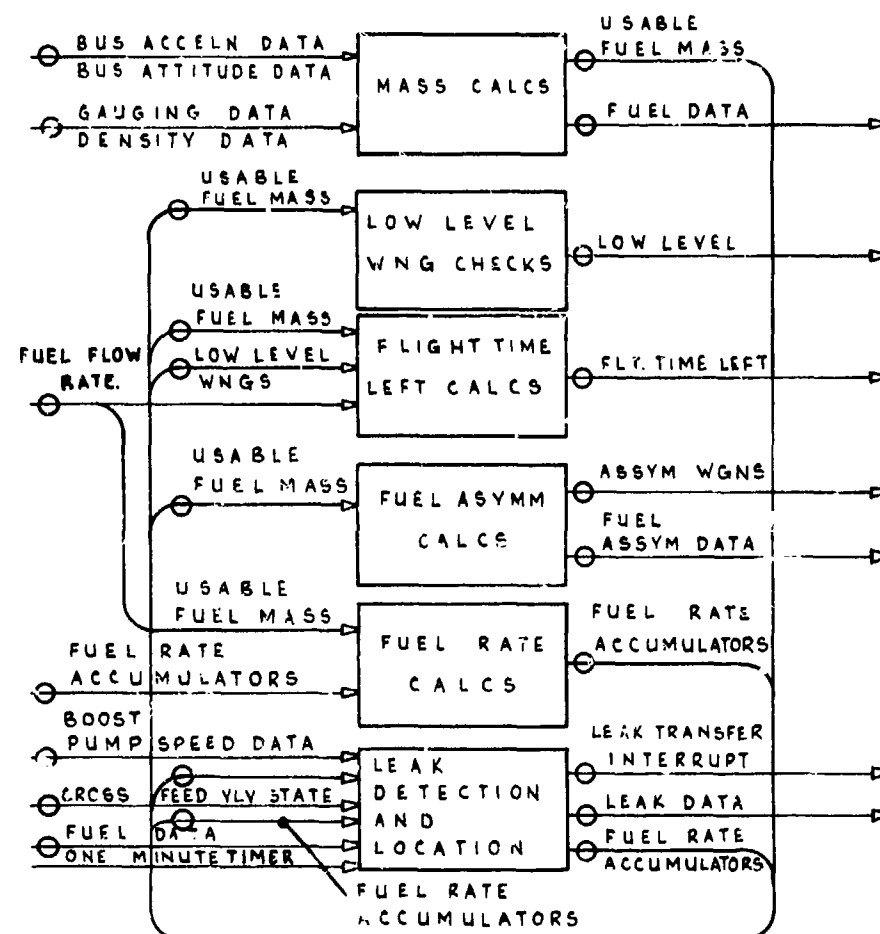


FIG. 13 SIMPLIFIED FMS OPERATIONAL DIAGRAM

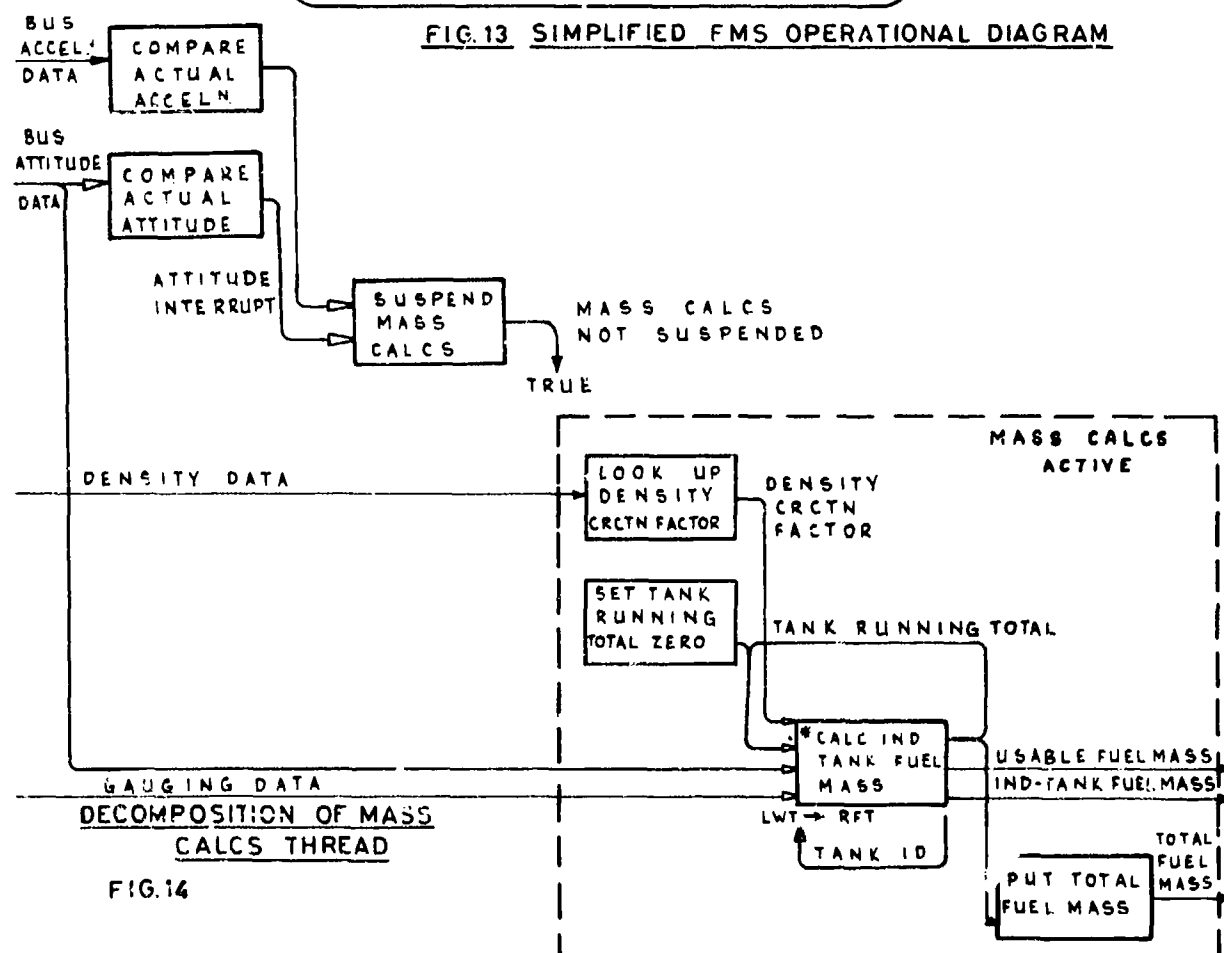


FIG. 14

A PEARL Softwaresystem for Multi-Processor Systems

Dr. P. Elzer
 Dr. H.-J. Schneider
 Dornier System GmbH
 Postfach 1360
 7990 Friedrichshafen
 F R G

Summary

Most today and all future systems will be processor based. There is a trend to multi-processor-systems. This is true for all types of systems, not excluding airborne ones. Up to now the majority of these systems is programmed in assembly language, a very awkward and expensive job.

Seeing the difficulties arising from low level coding, Dornier System implemented a High-Order-Language-System based on PEARL to program Multi-Processor-Systems in an airborne or similar environment. From this environment certain conditions for the implementation resulted. It was necessary to minimize the overhead produced by the operating system. The generated code was optimized to a very high efficiency with respect to time and memory.

Originally the aim of PEARL was process-control. Due to the application area here, sub-setting of PEARL was possible. This was done with high efficiency of code and a smaller modular operating system in mind.

On the other hand extensions to allow distributed processing were implemented.

The system consists of

- Lar (Subset of BASIC-PEARL)
- Compiler
- Assembler
- Linker/Loader
- Testing aids
- Special hardware for testing

It exists on a host-computer and is written in FORTRAN for portability. The target processors as implemented up to now are DORNIER DP 432, AEG 80-20 and DORNIER DP 426, which is based on an INTEL 8086.

The system was successfully used in several applications.

1. Introduction

It is a well known fact that High-Order Languages (HOL's) are one of the most successful means to improve the productivity of programmers as well as the quality of programs. For several years, however, there was a heated discussion among experts as to whether or not this was also true for real-time and other time-critical applications, like e.g. avionics or guidance and control applications. But mostly this discussion was not very well supported by quantitative data, and it was therefore felt necessary to conduct a study (1) on the applicability of High-Order Languages to guidance and control. The task was also, to find out, which special aspects had to be taken into consideration in this - admittedly difficult - application area. The study concentrated on the Language PEARL (= Process and Experiment Realtime Automation Language), because it was the most promising candidate language in the defense environment.

The results were very encouraging. It turned out that all of the relevant problems could be formulated in the language. It was not even necessary to exploit its full descriptive power. There was one exception, however: PEARL did not contain yet all the elements necessary for the programming of distributed systems and had therefore to be slightly expanded for this purpose.

Another important result was that the efficiency of the compiler and the size of the underlying operating system were of crucial importance for the usefulness of a HOL in guidance and control applications. The reasons for this are that, in this class of applications memory, however cheap, still is subject to severe limitations like physical size, energy consumption, or weight. Dynamic efficiency of the programs is of importance, too, because guidance and control processes tend to be extremely time-critical.

It also turned out that translators for HOL's in guidance and control had to provide very elaborate test and integration aids because of the intrinsic difficulties in testing and integrating embedded computer systems.

It was therefore decided that Dornier System should develop a PEARL translation system under contract with the German MOD (BMVg) which fulfilled the following requirements:

- Extreme Efficiency of the compiled code
- Elimination of Operating System Overhead as far as possible
- Possibility to program distributed systems
- Possibility to separate code-elements in RAM from those in PROM-type memory

- Optional support for system integration
- Adaptability to various target processors
- Easy transportability between host-processors

It was also obvious that it would not be sufficient to just develop a compiler. It was rather necessary to develop an entire PEARL translation system for distributed systems which consisted of the following components:

- Compiler-generator
- Compiler front-end
- Code generator
- Assembler
- Library management
- Modular operating system
- Linking loader
- Test and Integration aids

The construction principles of that system, and details about its implementation have already been published several times (3, 4, 5, 6).

2. The Language PEARL

The development and the properties of PEARL have also already been rather broadly published, e.g. in (7, 8). For the purposes of this paper it is therefore sufficient to concentrate on a few highlights.

2.1 Development and support of PEARL

PEARL was developed in the early seventies by a group of computer manufacturers, software houses and research institutes in the FRG. The development was organized by the University of Erlangen and mainly sponsored by the German Ministry of Research and Technology (BMFT). The first experimental compilers were finished in 1975 and full scale industrial applications started in 1977. Today, more than 200 PEARL-applications are in operation throughout the FRG in a broad variety of technological areas including defense systems.

Uniformity and continuity of PEARL are ensured by DIN standards. The draft standard DIN 66253, part 1, 'Basic PEARL', has been available since 1978. Part 2, 'Full PEARL', followed in August 1980. Besides, PEARL has been submitted to ISO for international standardization.

The support organization for PEARL is the 'PEARL-Association' with offices at the following addresses:

- PEARL Association
Graf-Rocke-Strasse 84
Postfach 1139
D-4000 Düsseldorf 1
F.R.G.
- PEARL Association
c/o Institut fuer Regelungstechnik und Prozessautomatisierung
Technical University of Stuttgart
Seidenstrasse 36
D-7000 Stuttgart 1
F.R.G.

2.2 Features of PEARL

PEARL has been developed for the application engineer. Great emphasis has therefore been laid upon language elements which facilitate the design of application programs in a real-time and process-control environment. The most important language elements belong to the following groups:

2.2.1 Real-time Language Elements:

To the knowledge of the authors PEARL contains currently the most complete set of elements for description and control of parallel processes. It is possible to declare program components as 'tasks' and initiate and control their execution as parallel processes, to react on interrupts and exceptions, and to connect these actions to external time conditions. E.g. it is possible to describe complex scheduling conditions like the following on language level:

```
AFTER 5 SEC ALL 7 SEC DURING 106 MIN
ACTIVATE MEASUREMENT PRIORITY 5;
```

This means that five seconds after the execution of this statement the computing process 'MEASUREMENT' is activated with priority five every seven seconds for a total period of one hundred and six minutes.

2.2.2 Description of the Hardware Configuration

In the 'system-division' of PEARL-programs the hardware configuration, especially the process peripherals and the data-paths, can be described separately from the application algorithm proper the 'problem division'. The relevant terminal points for I/O operations

can be named by symbolic identifiers and thus be referred to in the 'problem division' independently from the actual hardware. This capability greatly enhances documentation value and portability of PEARL programs.

2.2.3 Input/Output Language Elements

PEARL contains a consistent general I/O model for nonstandard devices as well as a set of user oriented I/O statements for the most usual operations. The general I/O model is based on the observation that each data-path in a digital system can be described by a sequence of 'data-stations' ('dations') and 'interfaces'. A data-station can either be a source of data, a sink, or intermediate storage. It further has so-called 'channels' which can be of the following types: 'data', 'control', 'signal' and 'interrupt'. The 'interfaces' are in principle sets of conversion routines which map the output characteristics of one dation onto the input characteristics of the following one.

The user-oriented I/O statements are the following ones:

- GET/PUT for character transfer
- READ/WRITE for file handling
- TAKE/SEND for process peripherals

All necessary format and control elements are provided.

2.2.4 Algorithmic Language Elements

Number and descriptive power of the language elements for the formulation of algorithms and procedures correspond to the state-of-the-art of modern programming languages. The concept of data types in Full-PEARL enables the user to define problem oriented, composite data types and new operators. These abstract data types permit a great number of compile-time checks and contribute to a refined modular structure.

2.2.5 Modular Program Structure

Last, but not least, PEARL supports modular program design and separate compilation of program components. A PEARL program is composed of separately compilable modules with exactly defined interfaces. This structure also greatly facilitates communication between the members of a project team and supports the modular composition of complex program systems.

3. The PEARL-Implementation by Dornier System

As already mentioned above, the characteristics of the PEARL-implementation by Dornier System are mainly dictated by the requirements of its application area. They are most obviously reflected in the choice of the implemented language subset.

3.1 The Language Subset

For the reasons mentioned above, those language elements were not implemented from which it was known that they would result in poor object code efficiency or unnecessary overhead at runtime.

In particular such elements are:

- Filehandling (on-board computers usually are not equipped with magnetic background storage devices)
- Formatting (on board there are practically no printing devices and the few which there are, can easily be handled by stream output of character strings)
- Absolute time (time is usually counted relative to 'mission start')
- Signals (exception handling is a source of huge overhead and it is mandatory that unplanned software conditions do not occur during the operational phase of a system)
- Structures (Application studies showed that measurement data are usually of homogeneous type).

On the other hand certain extensions had to be provided for the programming of distributed systems. However, it was a strict policy to keep them very small in order not to deviate too much from the original PEARL. Another important design criterium for these multi-computer extensions was that they had to be 'strategy independent', i.e. the user should be enabled to implement whatever concept be deemed optimal for the safety - or redundancy-strategy of his application. These considerations resulted in the following extensions:

- Declaration of entities with the attribute 'NET GLOBAL' of types 'variable', 'semaphore' and 'task'. These entities are then either copied into or made known to every processor in the distributed system.
- Operations on such entities. This was achieved without additional statements or operators, just by extending the semantics of existing operations (overloading).

Besides, there is a facility for the connection to 'external' tasks or procedures, which may e.g. be written in Assembler. Last, but not least, runtime checks can be inserted on a statement-by-statement basis by means of 'check/nocheck' statements.

3.2 The Compiler Front-End and its Technology

The technology, which had to be used for the translator, was determined by the requirements of adaptability to various target processors and easy transportability with respect to the host processor. This led to the usual separation into a 'front-end' which is independent of the target machine and translates PEARL into machine-independent intermediate code.

The compiler front-end is written in FORTRAN for the following reasons:

- FORTRAN translators are available for nearly every possible host computer
- A compiler, written in FORTRAN, is much more readable and much easier to maintain than any other one which is constructed according to an elaborate bootstrapping technology.

It turned out that this decision was the right one. The front-end could be adapted to the following host-computers with an effort of a few man-days each:

DEC PDP-11/70 and 11/44
 AEG-Telefunken 80-20/4
 Siemens 7760
 DEC PDP 10

Fig. 1 shows an overview over the structure of the entire translation system.

The intermediate representation had to be chosen according to the requirement of maximum code efficiency. Therefore it was not possible to use one of the usual virtual machine representations, because these usually do not contain any more all the information which was there in the source program and which is necessary for optimization. Besides, modern target processors usually have a more powerful instruction set than the one which happens to be implemented in a particular virtual machine architecture. This, too, leads to code-inefficiencies.

Therefore it was decided to use a completely target-independent intermediate representation, the so-called 'triple-code'. In principle it is a numeric representation of the program, where the individual operation is of the form:

operator, operand 1, operand 2

To sum up: the compiler front-end is written in FORTRAN and translates PEARL-Source programs into triple-code. It can detect approximately 200 different syntactical and semantical errors and identifies them by statement number, name of object and additional information, if necessary.

During translation the following listings can be produced on request:

- Source listing
- Cross-Reference listings for the following objects with their respective attributes (e.g. 'GLOBAL')
 - . Variables
 - . Tasks
 - . Semaphores
 - . Procedures
 - . Labels
 - . Datums
- Hierarchies of procedure calls
- Process hierarchy
- Synchronization structure
- Location of variables

3.3 The Code-generator

It produces symbolic assembly code with relative addresses for the target processor in question. This second intermediate layer has the disadvantage of an additional translation step, which may cost some time during translation, but this is more than balanced by the advantages. So, e.g. the assembler-listing provides an excellent means for final compiler testing and for easy linkage of external routines.

At the moment code-generators exist for the following target processors:

- DORNIER-MUDAS DP 432/433
- AEG-Telefunken 80-20
- DORNIER-MUDAS DP 426 (INTEL 80386-based)

3.4 Assembler

This component is necessary for the reasons given above. It is fully integrated into the translator system but usually adapted from the support software provided by the vendor of the target processor.

3.5 Pre-Linker

In case the linking-loader, which is provided by the vendor of the target processor, is not capable of handling the multi-module structure of PEARL-Programs, a pre-linker is provided, which performs the following functions:

- Identification of program modules to be linked together
- Distribution of code into RAM or ROM
- Distribution of program modules over the various processors of the distributed system
- Completeness check for the definition of global entities
- Linkeage of the operating system components required by the program
- Sorting of task-control-blocks and code segments
- Output of the control sequence for the linking loader

3.6 Linking-Loader

This tool performs the linkage process proper and produces absolute code. In case it cannot be taken from the vendor's software it is delivered together with the PEARL-System and is functionally integrated into the pre-linker.

3.7 Modular Operating System

This is a unique feature of the DORNIER PEARL-System. It allows efficient use of PEARL even in the smallest target configurations. This is achieved by abandoning the concept of an underlying, more or less autonomous and "monolithic" operating system. It is replaced by a set of routines which are automatically linked to the application program according to its requirements. These routines operate on task-control-blocks, time-order-blocks, etc. which are provided by the compiler. Thus it was possible to reduce the size of the operating system kernel to a mere 300 to 500 16-bit words, depending on the quality of the instruction set of the target processor. This kernel includes the following functions:

- Initialization
- Dispatcher
- An exit routine, which is executed if the system knows that there will be no task switching

The following functional modules can then be added automatically according to the requirements of the application program:

- Clock-routines
- Interrupt handler
- Activation of tasks
- Task-termination (regular)
- Task-termination (irregular; by 'TERMINATE')
- Suspension of tasks
- Continuation of suspended tasks
- Deletion of a schedule ('PREVENT')
- Inter-processor communication
- User command interface
- Character I/O ('GET', 'PUT')
- Procedure entry/exit
- Array indexing
- Arithmetic routines for FLOAT and DURATION types
- Comparison routines for FLOAT and DURATION types
- Type conversion routines
- Standard functions (ABS, SIGN)
- Handling of runtime errors

If all operating system services are invoked, it uses up to 4 to 6 K of 16-bit words, depending on the architecture of the target processor.

3.8 Library management

In order to be able to fully exploit the possibilities of the modular structure of PEARL programs and to enable the user to expand his system-library by himself, a special library management package is provided.

It contains the following functions:

- Inclusion of a new module
- Deletion of a module
- Listing of the Directory
- Modification of module names

3.9 Test and Integration Aids

Firstly, these include all the above mentioned listings which are produced by the compiler and serve as reference-documents for the user during test and integration.

Additionally there are runtime checks, which are on request inserted into the program either by the compiler or as operating system routines. The following errors can be monitored:

- Array index overflow
- Division by zero
- Range violation
- Conversion errors

These runtime checks can be enabled or disabled by the 'check/nocheck' feature.

Furthermore, several trace-routines can be built into the code:

- Jump trace
- Subroutine trace
- Call trace
- Task trace

Another important component is the debugger, which can be loaded together with the object program. It supports the following test functions:

- Activation and continuation of tasks
- Set and reset of breakpoints

- Output of environment information at breakpoints
- Input and display of values of variables
- Exit from Debugger (and return to normal execution of the program)

The design of this debugger allows for two modes of operation:

- Debugging on assembler level
- Debugging on source level

The first mode has already been implemented, the second one is being designed.

4. Application of the System

This PEARL Translator system has already been successfully used in several applications. Two of them are completed:

- A training simulator for the anti-aircraft tank 'Roland' (with 6 physically distributed processors)
- A gust alleviation system for a light aircraft

In both projects PEARL proved highly successful and the translator system fulfilled the expectations.

5. References

- 1/ H.-J. Schneider:
Modulare Software für Flugführung (Modular Software for Guidance and Control)
Dornier System, Report, June 1978
- 2/ DIN 66253, Part 1, preliminary standard
Programmiersprache PEARL, Basic PEARL
Beuth Verlag GmbH, Berlin, Köln, 1981
- 3/ H.-J. Schneider
PEARL-Softwaresystem für gekoppelte Klein- und Mikrorechner (PEARL-Software System for distributed Mini- and Microcomputers);
PEARL-Rundschau, Vol. 1, No 4, Dec. 1980 (pp 3-5)
- 4/ M. Ammann
PEARL für verteilte System (PEARL for distributed Systems),
Informatik-Fachberichte 39, 1981, Springer Verlag (pp 399-403)
- 5/ F. Graf
PEARL für Mikrocomputer (PEARL for microcomputers),
Informatik-Fachberichte 39, 1981, Springer Verlag (pp 413-421)
- 6/ M. Ammann, P. Elzer
Das PEARL-Uebersetzungssystem von Dornier System, Friedrichshafen
(The PEARL-Translator system by Dornier Systems, Friedrichshafen)
PEARL-Rundschau, Vol. 2, No 2, March 1981
- 7/ PEARL Subset for Avionic Applications; Agard Advisory Report No 90, Annex J,
(A Study of Standardization Methods for Digital Guidance and Control Systems),
May 1977
- 8/ T. Martin
PEARL at the Age of three; Proceedings of 4th IEEE Software Engineering Conference,
Sept. 1979 (pp 106-109)

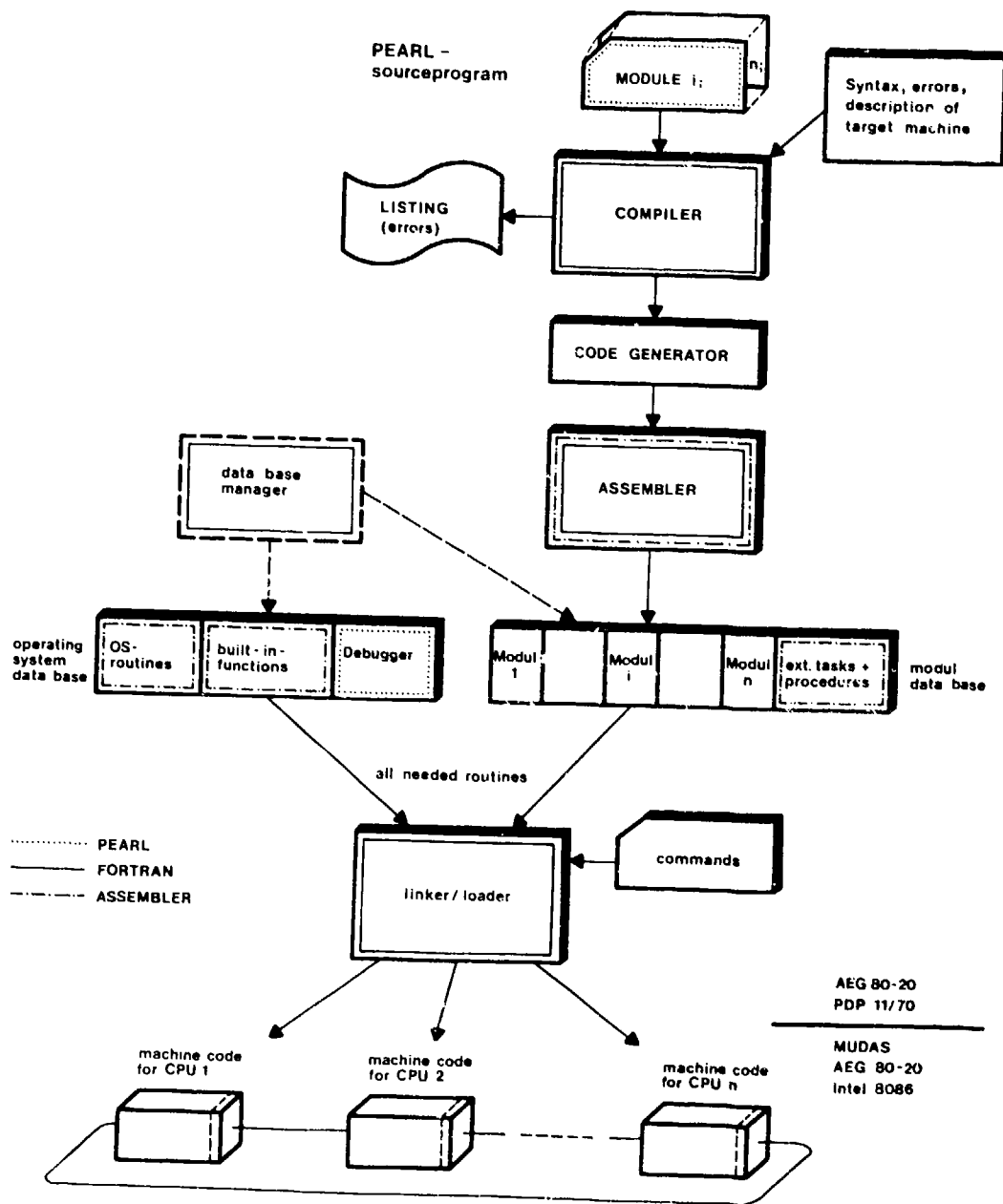


Fig 1 STRUCTURE OF THE SYSTEM

DISTRIBUTED AND DECENTRALIZED CONTROL IN FULLY DISTRIBUTED PROCESSING SYSTEMS

PHILIP H. ENSLOW JR.
GEORGIA INSTITUTE OF TECHNOLOGY
SCHOOL OF INFORMATION AND COMPUTER SCIENCE
ATLANTA, GEORGIA 30332

SUMMARY

Certainly one of the most important factors in designing and implementing fully distributed processing systems (FDPS) is the issue of distributed and decentralized control. Extremely loose coupling, both physical and logical, is an essential characteristic of an FDPS. This mode of organization and operation is quite different from the control of centralized systems. The first step in the development of distributed and decentralized control has been the examination of various models of control that may provide these features and the operational characteristics of those models.

1 FULLY DISTRIBUTED CONTROL

1.1 What is a Fully Distributed Processing System?

It has been determined that a high degree of both distribution and decentralization of control is essential if a system is to deliver a major proportion of those benefits being claimed for "distributed systems." Not only must the control be distributed, but the hardware and data must also exhibit similar characteristics. When all three system components, i.e., control, hardware, and data are sufficiently distributed, then the system can be characterized as "Fully Distributed." (See other paper in these proceedings [Ensl81] for a complete discussion of FDPS's.) This paper will focus on the control aspects of FDPS's.

1.2 Implications of the FDPS Definition on Control

1.2.1 General Nature of FDPS Executive Control

Several of the characteristics of an FDPS are found to directly impact the design and implementation of the executive control for such a system. These include system transparency to the user, extremely loose physical and logical coupling, and cooperative autonomy as the basic mode of component interaction. System transparency means that the FDPS appears to a user as a large uniprocessor which has available a variety of services. It must be possible for the user to obtain these services by naming them without specifying any information concerning the details of their physical location. The result is that system control is left with the task of locating all appropriate instances (copies) of a particular resource and choosing the instance to be utilized.

"Cooperative autonomy" is another characteristic of an FDPS heavily impacting its executive control. The "lower-level" control functions of both the logical and physical resource components of an FDPS are designed to operate in a "cooperatively autonomous" fashion. Thus, an executive control must be designed such that any resource is able to refuse a request even though it may have physically accepted the message containing that request. Degeneration into total anarchy is prevented by the establishment of a common set of criteria to be followed by all resources in determining whether a request is accepted and serviced as originally presented, accepted only after bidding/negotiation, or rejected.

Another important FDPS characteristic that definitely affects the design of its executive control is the extremely loose coupling of both physical and logical resources. The components of an FDPS are connected by communication paths of relatively low bandwidth. The direct sharing of primary memory between processors is not acceptable. Even though the logical coupling could still be loose with this physical interconnection mechanism, the presence of a single critical hardware element, the shared memory would create fault-tolerance limitations. All communication takes place over "standard" input/output paths. The actual data rates that can be supported are primarily a function of the distance between processors and the design of their input/output paths. In any event, the transfer rates possible will probably be much less than memory transfer rates. This implies that the sharing of information among components on different processors is greatly curtailed, and system control is forced to work with information that is usually out-of-date and, as a result, inaccurate.

The control of an FDPS requires the action and cooperation of components at all layers of the system. This means that there are elements of FDPS control present in the lowest levels of the hardware as well as software components. This paper is primarily interested in the software components of the FDPS control which are typically referred to as the "executive control."

The executive control is responsible for managing the physical and logical resources of a system. It accepts user requests and obtains and schedules the resources necessary to satisfy a user's needs. As mentioned earlier, these tasks are accomplished so as to unify the distributed components of the system into a whole and provide system transparency to the user.

1.2.2 Why Not Centralized Control?

Why then is a centralized method of control not appropriate? In systems utilizing a centralized executive control, all of the control processes share a single coherent and deterministic view of the entire system state. An FDPS, though, contains only loosely-coupled components, and the communication among these components is restricted and subject to variable time delays. This means that one cannot guarantee that all processes will have the same view of the system state [Jens78]. In fact, it is an important characteristic of an FDPS that they will not have a consistent view.

A centralized executive control weakens the fault-tolerance of the overall system due to the existence of a single critical element, the executive control itself. This obstacle, though, is not insurmountable for strategies do exist for providing fault-tolerance in centralized applications. Garcia-Molina [Garco79], for example, has described a scheme for providing fault-tolerance in a distributed data base management system with a centralized control. Approaches of this type typically assume that failures are extremely rare events and that the system can tolerate the dedication of a relatively long interval of time to reconfiguration. These restrictions are usually unacceptable in an FDPS environment where it is important to provide fault-tolerance with a minimum of disruption to the services being supported.

Also, the extremely important issue of overall system performance must be considered. A distributed processing system is expected to utilize a large quantity and a wide variety of resources. If a completely centralized executive control is implemented, there is a high probability that a bottleneck will be created in the node executing the control functions. A distributed and decentralized approach to control attempts to remove this bottleneck by dispersing the control decisions among multiple components on different nodes.

1.2.3 Distributed vs. Decentralized

This paper advocates utilizing an approach for the control of an FDPS that is both distributed and decentralized. There is a clear distinction between the terms "distributed" and "decentralized" as they are used in the context of this project. "Distributed control" is characterized by having its executing components physically located on different nodes. This means there are multiple loci of control activity. In "decentralized control," on the other hand, control decisions are made independently by separate components at different locations. In other words, there are multiple loci of control decision making. Thus, distributed and decentralized control has active components located on different nodes and those components are capable of making independent control decisions.

2. ISSUES IN DISTRIBUTED CONTROL

Before examining specific aspects of executive control in an FDPS, a look at some of the various issues of distributed control is appropriate. There are three primary issues that require examination: 1) the effect of the dynamics of FDPS operation on an executive control, 2) the nature of the information an executive control must maintain, and 3) the principles to be utilized in the design of an executive control.

2.1 Dynamics

Dynamics are an inherent characteristic of the operation of an FDPS. They are found in the work load presented to the system, the availability of resources, and the individual work requests submitted. The dynamic nature of each of these provides the FDPS executive control with many unique problems.

2.1.1 Workload Presented to the System

In an FDPS, work requests can be generated either by users or active processes and can originate at any node. Such work requests potentially can require the use of resources on any processor. Thus, the collection of executive control procedures must be able to respond to requests arriving at a variety of locations from a variety of sources. Each request may require system resources located on one or more nodes, not necessarily including the originating node. One of the goals of an FDPS executive control is to respond to these requests in a manner such that the load on the entire system is balanced.

2.1.2 Availability of Resources

Another dynamic aspect of the FDPS environment concerns the availability of resources within the system. As mentioned above, a request for a service to be provided by a system resource may originate at any location in the system. In addition, there may be multiple copies of a resource or possibly multiple resources that provide the same functionality (e.g., there may be functionally equivalent FORTRAN compilers available on several different nodes). Since resources are not immune to failures, the possibility of losing existing resources or gaining both new and old resources exists. Therefore, an FDPS executive control must be able to manage system resources in a dynamic environment in which the availability of a resource is unpredictable.

2.1.3 Individual Work Requests

Finally, the dynamic nature of the individual work requests must be considered. As mentioned above, these work requests define, either directly or indirectly, a set of cooperating processes which are to be invoked. An indirect definition of the work to be done occurs when the work request is itself the name of a command file or contains the name of a command file in addition to names of executable files or directly executable statements. A command file contains a collection of work requests formulated in command language statements (see Figure 1 for a description of the syntax for a suitable command language) that are interpreted and executed when the command file is invoked. The concept of a command file is similar to that of a procedure file which is available on several current systems.

Management of the processes for a work request thus includes the possibility that one or more of the processes are command files requiring command interpretation. The presence of command files will also result in the inclusion of additional information in the task graph or possibly additional task graphs.

An important objective of work request management is to control the set of processes and do so in such a manner that the inherent parallelism present in the operations to be performed is exploited to the maximum. In addition, situations in which one or more of the processes fail must also be handled.

2.2 Information

All types of executive control systems require information in order to function and perform their mission. The characteristics of the information available to the executive control is one aspect of fully distributed systems that result in the somewhat unique control problems that follow:

1. Because of the nature of the interconnection links and the delays inherent in any communication process, system information on hand is always out of date.
2. Because of the autonomous nature of operation of all components, each processor can make "its own decision" as how to reply to an inquiry; therefore, there is always the possibility that information received is incomplete and/or inaccurate.
3. Because of the inherent time delays experienced in exchanging information among processes on different nodes, some information held by two processes may conflict during a particular time interval.

2.3 Design Principles

Designing the system control functions required for the extremely loosely-coupled environment of an FDPS and implementing those functions to operate in that environment will certainly require the application of some new design principles in addition to those commonly utilized in operating systems for centralized systems. These design principles must address at least the two distinguishing characteristics of FDPS's:

- System information available, and
- Nature of resource control

2.3.1 System Information

The various functions of an FDPS executive control must be designed recognizing that system information is:

- "Expensive" to obtain
- Never fully up-to-date
- Usually incomplete
- Often inaccurate

All of these characteristics of system information result from the fact that the components providing the information are interconnected by relatively narrow bandwidth communication paths and that those components are operating somewhat autonomously with the possibility that their state may change immediately after a status report has been transmitted. Further, it is important to note that the mere existence (or disappearance) of a resource is not of interest to a specific component of the FDPS executive control until that component needs that information.

The design principles applying to system information that have been identified thus far include the following:

1. Economy of communication: ask for only the information required.
2. Resiliency: be prepared to recover and continue in the absence of replies.
3. Flexibility: be prepared to recover and continue if the information provided proves to be inaccurate when it is utilized.

2.3.2 Resource Control

Since all of the resources are operating under local control under the policies of cooperative autonomy, all requests for service, or the utilization of any resource such as a file, must be effected through negotiations that culminate in positive acknowledgements by the server. In all instances, the control function requesting a service or a resource must be prepared for refusal.

3. CHARACTERIZATION OF FDPS WORK REQUESTS

3.1 The Work Request

One of the goals of an FDPS is the ability to provide a hospitable environment for solving problems that allows the user to utilize the natural distribution of data to obtain a solution which may take the form of an algorithm consisting of concurrent processes. The expression of the solution is in terms of a work request that describes a series of cooperating processes, the connectivity of these processes (how the processes communicate), and the data files utilized by these processes. This description involves only logical entities and does not contain any node-specific information. A description of one command language capable of expressing requests for work in this fashion can be found in [Akin78] (see Figure 1).

3.2 Impact of the Work Request on the Control

The nature of allowable work requests (not just the syntax but what can actually be accomplished via the work request) determines to a large extent the functionality of an executive control. Therefore, it is important to examine the characteristics of work requests and further to see how variations in these characteristics impact the strategies utilized by an FDPS executive control.

Five basic characteristics of work requests have been identified:

1. the external visibility of references to resources required by the task,
2. the presence of any interprocess communication (IPC) specifications,
3. the number of concurrent processes,
4. the nature of the connectivity of processes, and
5. the presence of command files.

3.2.1 Visibility of References to Resources

References to the resources required to satisfy a work request may either be visible prior to the execution of a process associated with the work request or embedded in such a manner that some part of the work request must be executed to reveal the reference to a particular resource. A resource is made "visible" either by the explicit statement of the reference in the work request or through a declaration associated with one of the resources referenced in the work request. An example of the latter means of visibility is a file system in which external references made from a particular file are identified and stored in the "header" portion of the file. In this case, the identity of a reference can be obtained by simply accessing the header.

The greatest impact of the visibility characteristic of resource requirements occurs in the construction of task graphs and the distribution of work. The time at which resource requirements are detected and resolved determines when and how parts of the task graph can be constructed. Similarly, some work cannot be distributed until certain details are resolved. For example, consider a case where resource references cannot be resolved until execution time. Assume there exist two processes X and Y where process X has a hidden reference to process Y. An executive control cannot consider Y in the work distribution decision that is made in order to begin execution of X. The significance of this is that certain work distribution decisions may not be "globally optimal" because total information was not available at the time the decision was made.

3.2.2 The Number of Concurrent Processes

A work request can either specify the need to execute only a single process or the execution of multiple processes which may possibly be executed concurrently. Obviously with multiple processes, more resource availability information must be maintained; and there is a corresponding increase in the data to the work distribution and work allocation phases of control. In addition, the complexity of the work distribution decision algorithm increases with more resources needing to be allocated and multiple processes needing scheduling. The complexity of controlling the execution of the work request is also increased with the presence of multiple processes since the control must monitor multiple processes for each work request.

3.2.3 The Presence of Interprocess Communication

The problems described in the previous paragraph are amplified by the presence of communication connections between processes. When interprocess communication is described in a work request, the work distribution decision must consider the requirement for communication links. In addition, a compromise must be made in order to satisfy the conflicting goals of maximizing the inherent parallelism of the processes of the work request and minimizing the cost of communication among these processes. The control activity required during execution is also impacted by the presence of interprocess communication. It must provide the means for passing messages, buffering messages, and providing synchronization to insure that a reader does not underflow and a writer does not overflow the message buffers.

3.2.4 The Nature of Process Connectivity

There are a variety of techniques available for expressing interprocess communication including pipes (see [Ritch78]) and ports (see [Balz71, Hare78, Suns77, Zuck77]). There are a number of approaches to realizing these different forms of interprocess communication. The main impact on an executive control, though, is in those components controlling process execution.

3.2.5 The Presence of Command Files

A command file is composed of work requests. Execution of a work request that references a command file results in a new issue dealing with the construction of task graphs. This issue is concerned with whether a new task graph should be constructed to describe the new work request or should these new processes be included in the old task graph. The difference between these two approaches becomes important during work distribution. It is assumed that the work distribution decision will be made only with the information available in the task graph. Thus, with the first approach, only those tasks in the new work request are considered while the second approach provides the ability to take into consideration the assignment of tasks from previous work requests.

3.3 A Classification of Work Requests

This examination of the characteristics of FDPS work requests has led to the identification of five basic attributes which have significant impact on an executive control. In Figure 2, all possible types of work requests are enumerated resulting in 32 different forms of work requests. It should be noted, though, that 16 of these (those with an asterisk beside the task number) contain conflicting characteristics and thus are impossible.

4. CHARACTERISTICS OF FDPS CONTROL MODELS

4.1 Approaches to Implementing FDPS Executive Control

There are two basically different approaches available for implementing an operating system for a distributed processing system, the base-level approach and the meta-system approach [Thom78]. The base-level approach does not utilize any existing software and, therefore, requires the development of all new software. This includes software for all local control functions such as memory management and process management. In contrast, the meta-system approach utilizes the "existing" operating systems (called local operating systems (LOS)) from each of the nodes of the system. Each LOS is "interfaced" to the distributed system by a network operating system (NOS) which is designed to provide high level services available on a system-wide basis. The meta-system approach is usually preferred due to the availability of existing software to accomplish local management functions, thus, reducing development costs [Thom78].

Figure 3 depicts a logical model applicable to an FDPS executive control utilizing either approach. The LOS handles the low-level (processor-specific) operations required to directly interface with users and resources. In the meta-system approach, the LOS represents primarily the operating systems presently available for nodes configured in stand-alone environments. The LOS resulting from a base-level approach has similar functionality; however, it represents a new design, and certain features may be modified in order to allow the NOS to provide certain functions normally provided by the LOS. Any "network" operations are performed by the NOS. System unification is realized through the interaction of NOS components, possibly residing on different processors, acting in cooperation with appropriate LOS components. Communication among the components is provided by the message handler which utilizes the message transport services.

4.2 Information Requirements

Two types of information are required by an executive control, information concerning the structure of the set of tasks required to satisfy the work request and information about system resources. This data is maintained in a variety of data structures by a number of different components.

4.2.1 Information Requirements for Work Requests

Each work request identifies a set of cooperating tasks, nodes in a logical network that cooperate in execution to satisfy a request and the connectivity of those nodes. Figure 1 illustrates the notation used in this project to express work requests. An example of a work request using this notation is presented in Figure 4. Work requests as linear textual forms can be easily accepted and manipulated by the computer system; however, task graphs, which are an internal control structure used to describe work requests, must be represented in a manner such that the linkage information is readily available. This can take the form of the explicit linking of node control blocks (Figure 5) or an interconnection matrix (Figure 6).

Information concerning a particular task, i.e., logical node, is maintained in a node control block (Figure 5). Associated with each logical node is an execution file, a series of input files, and a series of output files. The node control block contains information on each of these entities that includes the name of the resource, the locations of possible candidates that might provide the desired resource, and the location of the candidate resource chosen to be utilized in the satisfaction of the work request. In addition to this information, the node control block maintains a description of all interprocess communication (IPC) in which the node is a party. This consists of a list of input ports and output ports. (Interprocess communication is a term describing the exchange of messages between cooperating processes of a work request.) Typically, a message is "sent" when it is written to the output port of a process. The message is then available for consumption by any process possessing an input port that is connected to the previously mentioned output port. The message is actually consumed or accepted when the process owning the connected input port executes a READ on that port.

A global view of interprocess communication is provided by the node interconnection matrix (Figure 6). This structure indicates the presence or absence of an IPC link between an output port of one node and an input port of another node. Thus, links are assumed to carry data in only a single direction.

An example of a task graph resulting from the work request in Figure 4 utilizing the direct linking of node control blocks is presented in Figure 7. Figure 8 illustrates the utilization of an interconnection matrix.

4.2.2 Information Requirements for System Resources

Regardless of how the executive control is realized (i.e., how the components of the executive control are distributed and how the control decisions are decentralized), information concerning all system resources (processors, communication lines, files, and peripheral devices) must be maintained. This information includes at a minimum an indication of the availability of resources (available, reserved, or assigned). Preemptable resources (e.g., processors and communication lines) capable of accommodating more than one user at a time may also have associated with them utilization information designed to guide an executive control in its effort to perform load balancing.

As discussed below, there are a number of techniques that may be employed to gather and/or maintain the system resource information.

4.3 Basic Operations of FDPS Control

The primary task of an executive control is to process work requests that can best be described as logical networks. A node of a logical network specifies an execution file that may either contain object code or commands (work requests), input files, and output files. These files may reside on one or more physical nodes of the system and there may be multiple copies of the same file available. Thus, to process a work request, an FDPS executive control must perform three basic operations: 1) gather information, 2) distribute the work and allocate resources, and 3) initiate and monitor task execution. These operations need not be executed in a purely serial fashion but may take a more complex form with executive control operations executed simultaneously or concurrently with task execution as the need arises.

Examination of the basic operations in further detail (Figure 9) reveals some of the variations possible in the handling of work requests. Two steps exist in information gathering --- 1) collecting information about task requirements for the work request and 2) identifying the resources available for satisfying the request requirements. Information gathering is followed by the task of distributing the work and allocating resources. If this operation is not successful, three alternatives are available. First, more information on resource availability can be gathered in an attempt to formulate a new work distribution. There may have been a change in the status of some resources since the original request for availability information. Second, more information can be gathered as above, but this time the requester will indicate a willingness to "pay more" for the resources. This is referred to as bidding to a higher level. Finally, the user can simply be informed that it is impossible to satisfy his work request.

4.3.1 Information Gathering

Upon receiving a work request, the first task of the control is to discover what resources are needed to satisfy the work request (Figure 10) and which resources are available to fill these needs (Figure 11). Each work request includes a description of a series of tasks and the connectivity of those tasks. Associated with each task is a series of files. One is distinguished as the execution file and the rest are input/output files. The executive control must first determine which files are needed. It then must examine each of the execution files to determine the nature of its contents (executable code or commands). Each task will need a processor resource(s), and those tasks containing command files will also require a command interpreter.

An FDPS executive control must also determine which of the system resources are available. For nonpreemptable resources, the status of a resource can be either "available," "reserved," or "assigned." A reservation indicates that a resource may be used in the future and that it should not be given to another user. Typically, there is a time-out associated with a reservation that results in the automatic release of the reservation if an assignment is not made within a specified time interval. The idea here is to free resources that otherwise would have been left unavailable by a lost process. The process may be lost because it failed, its processor failed, or the communication link to the node housing the particular resource may have failed. An assignment, on the other hand, indicates that a resource is dedicated to a user until the user explicitly releases that assignment. Preemptable resources may be accessed by more than one concurrent user and thus can be treated in a different manner. For these resources, the status may be indicated by more continuous values (e.g., the utilization of the resource) rather than the discrete values described above.

4.3.2 Work Distribution and Resource Allocation

The FDPS executive control must determine the work distribution and the allocation of system resources (Figure 12 & 13). This process involves choosing from the available resources those that are to be utilized. This decision is designed to achieve several goals such as load balancing, maximum throughput, and minimum response time. It can be viewed as an optimization problem similar in many respects to that discussed by Morgan [Morg77].

Once an allocation has been determined, the chosen resources are allocated and the processes comprising the task set are scheduled and initiated. If a process cannot be immediately scheduled, it may be queued and scheduled at a later time. When it is scheduled, a process control block and any other execution-time data structures must be created.

4.3.3 Information Recording

Information is recorded as a result of management actions as well as providing a means to maintain a historical record or audit trail of system activity. The information recording resulting from management actions maintains the system state and provides information for decision making. The historical information is useful in monitoring system security. It provides a means to examine past activity on a system in order to determine if a breach of security occurred or how a particular problem or breach of security may have occurred.

Management information is maintained in various structures, including the task graph. The task graph is used to maintain information about the structure of an individual work request, and, thus, its contents change as progress on the work request proceeds. A task graph is created when a work request is first discovered, and information is then constantly entered into the structure as work progresses through information gathering to work distribution and resource allocation and on to task execution. The task graph remains active until completion of the work request.

Much of the information contained in the task graph is applicable to historical records. In fact, the task graph can be used to house historical information as it is gathered during work request processing. Upon completion of the work request, the historical information is extracted and entered into the permanent historical file. Alternatively, the historical file can be created directly skipping the intermediate task graph structure.

4.3.4 Task Execution

Finally, an executive control must monitor the execution of active processes. This includes providing interprocess communication, handling requests from active processes, and supervising process termination. The activities associated with interprocess communication include establishing communication paths, buffering messages, and synchronizing communicating processes. The latter activity is necessary to protect the system from processes that flood the system with messages before another process has time to absorb the messages. Active processes may also make requests to the executive control. These may take the form of additional work requests or requests for additional resources. Work requests may originate from either command files or files containing executable code.

An executive control must also detect the termination of processes. This includes both normal and abnormal termination. After detecting process termination, it must inform processes needing this information that termination has occurred, open files must be closed, and other loose ends must be cleaned up. Finally, when the last process of a work request has terminated, it must inform the originator of the request of the completion of the request.

4.3.5 Fault Recovery

If portions (tasks) of the work request are being performed on different processors, there is inherently a certain degree of fault recovery possible. The problem is in exploiting that capability. The ability to utilize "good" work remaining after the failure of one or more of the processors executing a work request depends on the recovery agent having knowledge of the location of that work and the ability of the recovery agent to reestablish the appropriate linkages to the new locations for the portions of the work that were being executed on the failed processor(s).

5. VARIATIONS IN FDPs CONTROL MODELS

There is an extremely large number of features by which variations in distributed control models can be characterized. Of these, only a few basic attributes appear to deserve attention. These include the nature of how and when a task graph is constructed, the maintenance of resource availability information, the allocation of resources, process initiation, and process monitoring. In this section, these issues are examined; but again, since the number of variations possible in each issue is rather large, only those choices considered significant are discussed. Table 2 contains a summary of the problems that have been identified and possible solutions (significant and reasonable solutions) to these problems.

5.1 Task Graph Construction

The task graph is a data structure used to maintain information about the applicable task set. The nodes of a task graph represent the tasks of the task set, and the arcs represent the connectivity or flow of information between tasks. There are basically four issues in task graph construction: 1) who builds a task graph, 2) what is the basic structure of a task graph, 3) where are the copies of a task graph stored, and 4) when is a task graph built.

The identity of the component or components constructing the task graph is an issue that presents three basic choices. First, a central node can be responsible for the construction of task graphs for all work requests. Another choice utilizes the control component as the node receiving the work request to construct the task graph. Finally, the job building task graph can be distributed among several components. In particular, the nodes involved in executing individual tasks of the work request can be responsible for constructing those parts of the task graph that they are processing.

The general nature of the task graph itself provides two alternatives for the design of an executive control. What is of concern here is not the content of a task graph but rather its basic structure. One alternative is to maintain a task graph in a single structure regardless of how execution is distributed. The other is to maintain the task graph as a collection of subgraphs with each subgraph representing a particular work request. For example, a subgraph can represent that portion of the work request that is executed on a particular node at which that subgraph is stored.

Another issue in task graph construction concerns where the various copies of the task graph are stored. If the control maintains a task graph as a unified structure representing the complete set of tasks for a work request, this structure may either be stored on a single node, or redundant copies can be stored on multiple nodes. The single node can either be a general node that is used to store all task graphs, the node at which the original work request arrived (the source node), or a node chosen for its ability to provide this work request with optimal service. If the task graph is divided into several subgraphs, these can be distributed on multiple nodes.

Finally, there is the issue concerning the timing of task graph construction in the sequence of steps to deal with a work request processing. Two choices are available: 1) the task graph can be constructed completely, at least to the maximum extent possible, before execution is begun, or 2) the task graph can be constructed incrementally as execution progresses.

5.2 Resource Availability Information

Another possible source of variability for control models is the maintenance of resource availability information. What is of importance here is "Who maintains this information" and "Where is this information maintained." A particular model need not uniformly apply the same technique for maintaining resource availability information to all resources. Rather, the technique best suited to a particular resource class may be utilized.

The responsibility for maintaining resource availability information can be delegated in a variety of ways. The centralized approach involves assigning a single component this responsibility. In this situation, requests and releases for resources flow through the specialized component which maintains the complete resource availability information in one location.

Another variation of this technique maintains complete copies of the resource availability information at several locations [Cah79a,b]. Components at each of these locations are responsible for updating their copy of the resource availability information in order to keep it consistent with the other copies. This requires a protocol to insure that consistency is maintained. For example, two components should not release a resource for writing to different users at the same time. To provide this control, messages containing updates for the information tables must be exchanged among the components. In addition, a strategy for synchronizing the release of resources is required. An example of such a strategy is found in [Cah79a,b] where a baton is passed around the network. The holder of the baton is permitted to release resources.

Another approach exhibiting more decentralization requires dividing the collection of resources into subsets or classes and assigning separate components to each subset. Each component is responsible for maintaining resource availability information on a particular subset. In this case, requests for resources can only be serviced by the control component responsible for that resource. Resources may be released in a manner such that the desired manager is readily identifiable. Alternatively, a search may be required in order to locate the appropriate manager. This search may involve passing the request from component to component until one is found that is capable of performing the desired operation.

Preemptable resources which can be shared by multiple concurrent users (e.g., processors and communication lines) do not necessarily require the maintenance of precise availability information. For these resources it is reasonable to maintain only approximate availability information because such

5.3 Allocating Resources

One of the major problems experienced in the allocation of resources is concurrency control. In a hospitable environment, it is possible to ignore concurrency control. The users are given the responsibility of insuring that access to a shared resource such as a file is handled in a consistent manner. In other environments, for example that presented by an FDPS, this is an important issue. In an FDPS, the problem is even more difficult than in a centralized system due to the loose coupling inherent in the system.

There are basically two approaches to solving the problem of concurrent requests for shared resources. The first utilizes the concept of a reservation. Prior to the allocation of resources (possibly when resource availability information is acquired), a resource may be reserved. The reservation is effective for only a limited period (a period long enough to make a work distribution decision and allocate the resources determined by the decision) and prevents other users from acquiring the resource. The other solution to this problem is to make the work distribution decision without the aid of reservations. If resources cannot be allocated, the executive control will either wait until they can be allocated or attempt a new work distribution.

5.4 Process Initiation

Several issues arise concerning process initiation. Chief among these is the distribution of responsibility. There are a large number of organizations possible, but only a few are reasonable. The basic organizations utilize either a single manager, a hierarchy of managers, or a collection of autonomous managers. Two approaches result from the single manager concept. In the first organization, a central component is in charge of all work requests and the processes resulting from these work requests. All decisions concerning the fate of processes and work requests are made by this component. A variation on this organization assigns responsibility at the level of work requests. In other words, separate components are assigned to each work request. Each component makes all decisions concerning the fate of a particular work request and its processes.

Management can also be organized in a hierarchical manner. There are a variety of ways hierarchical management can be realized, but we will concentrate on only two, the two-level hierarchy and the n-level hierarchy. The two-level hierarchy has at the top level a component that is responsible for an entire work request. At the lower level are a series of components each responsible for an individual task of the work request. The lower level components take direction from the high level component and provide results to this component. The n-level hierarchy utilizes in its top and bottom levels the components described for the two-level hierarchy. The middle levels are occupied by components that are each responsible for a subgraph of the entire task graph. Therefore, a middle component takes direction from and reports to a higher level component which is in charge of a part of the task graph that includes the subgraph for which the middle component is responsible. The middle component also directs lower level components each of which are responsible for a particular task.

Another organizational approach utilizes a series of autonomous management components. Each component is in charge of some subset of the tasks of a work request. Cooperation of the components is required in order to realize the orderly completion of a work request.

Regardless of the organization, at some point, a request for the assumption of responsibility by a component will be made. Such a request may be reasonably denied for two reasons: 1) the component does not possess enough resources to satisfy the request (e.g., there may not be enough space to place a new process on an input queue), or 2) the component may not be functioning. The question that arises concerns how this denial is handled. One solution is to keep trying the request either until it is accepted or until a certain number of attempts have failed. In this case if the request is never accepted, the work request is abandoned, and the user is notified of the failure. Instead of abandoning the work request, it is possible that a new work distribution decision can be formulated utilizing the additional knowledge concerning the failure of a certain component to accept a previous request.

5.5 Process Monitoring

The task of monitoring process execution presents the FDPS executive control with two major problems, providing interprocess communication and responding to additional work requests and requests for additional resources. With regard to the problem of interprocess communication, there is some question as to the nature of the communication primitives an FDPS executive control should provide. This question arises due to the variety of communication techniques being offered by current languages. There are two basic approaches found in current languages, synchronized communication and unsynchronized communication (buffered messages). Synchronized communication requires that the execution of both the sender and the receiver be interrupted until a message has been successfully transferred. Examples of languages utilizing this form of communication are Hare's Communicating Sequential Processes [Hoar78] and Brinch Hansen's Distributed Processes [Brin78]. In contrast, buffered messages allow the asynchronous operation of both senders and receivers. Examples of languages using this form of communication are PLITS [Feld79] and STARMOD [Cook80].

The executive control is required to provide communication primitives that are suitable to one of the communication techniques discussed above. If the basic communication system utilizes synchronized communication, both techniques can be easily handled. The problem with this approach is that there is extra overhead incurred when providing the message buffering technique. On the other hand if the basic communication system utilizes unsynchronized communication, there will be great difficulty in realizing a synchronized form of communication.

The task of monitoring processes also involves responding to requests generated by the executing tasks. These may be either requests for additional resources (e.g., an additional file) or new work requests. If the request is a work request, there is a question as to how a new set of tasks is to be handled. It should either be included in the existing task

5.6 Process Termination

When a process terminates there is always some cleanup work that must be accomplished (e.g., closing files, returning memory space, and deleting records concerning that process from the executive control's work space). In addition, depending on the reason for termination (normal or abnormal), other control components may need to be informed of the termination. In the case of a failure, the task graph will contain the information needed to perform cleanup operations (e.g., the identities of the processes needing information concerning the failure). Both the nature of the cleanup and the identity of the control components that must be informed of the termination are determined from the design decisions resulting from the issues discussed above.

5.7 Examples

To gain a better appreciation of some of the basic issues of control in an FPPS, it is useful to examine an example of work request processing on an FPPS. In the example, emphasis is placed on the operations involved in the construction of task graphs. The work distribution decision that is utilized is a simple one that assigns the execution of processes to the same nodes that house the files containing their code. The primary concern of this example (Figure 4) is the impact of variations in work requests on task graph construction. In this example, the various parts of the overall task graph describing the complete work request are stored on the nodes utilized by each part. Other techniques for storing the task graphs may also be utilized. In the example, the following symbols are utilized:

[]	visible external reference(s)
{ }	embedded external reference(s)
(n)A	responsibility for A delegated from node n
A(n)	responsibility for A delegated to node n
a-->b	IPC from process a to process b
A,B,...	uppercase letters indicate command files
a,b,...	lowercase letters indicate executable files
u,v,w,x,y,z	indicate data files

Now that we have taken a look at the construction of task graphs in a broad sense, let us examine the details of the task of processing a work request. This is illustrated in two figures. Figure 15 outlines the basic steps involved in work request processing. Finally, Figure 16 depicts the steps involved in processing a specific work request. In this case, the work request is the same as that examined in the example of task graph building (Figure 14).

6. CONCLUSIONS

Thus far it has been possible to identify a number of the characteristics of a distributed and decentralized control system and to identify some of its operational features. The evaluation of this mode of system control is the next task.

7. ACKNOWLEDGEMENTS

Much of the work reported on here has been performed by Timothy G. Saponas as part of his research work for the Ph.D. degree. His area of primary interest is distributed and decentralized control. The work has been performed as part of the Georgia Institute of Technology Research Program in Fully Distributed Processing Systems. The support for this specific project was provided by the Department of the Air Force, Rome Air Development Center, Griffiss Air Force Base, New York, under contract F30602-78-C-0120.

8. REFERENCES

- Akin78 Akin, T. Allen, Flinn, Perry B., Forsyth, Daniel H., "A Prototype for an Advanced Command Language," Proceedings of the 16th Annual Southeastern Regional ACM Conference (April, 1978): 96-102.
- Balz71 Balzer, R. M., "PORTS - A Method for Dynamic Interprogram Communication and Job Control," AFIPS Conference Proceedings 38 (1971 Spring Joint Computer Conference): 485-439.
- Brin78 Brinch Hansen, Per, "Distributed Processes: A Concurrent Programming Concept," Communications of the ACM 21 (November, 1978): 934-941.
- Caba79a Cabanel, J. P., Marouane, M. N., Besbes, R., Sazbon, R. D., and Diarra, A. K., "A Decentralized OS Model for ARAMIS Distributed Computer System," Proceedings of the First International Conference on Distributed Computing Systems (October, 1979): 529-535.
- Caba79b Cabanel, J. P., Sazbon, R. D., Diarra, A. K., Marouane, M. N., and Besbes, R., "A Decentralized Control Method in a Distributed System," Proceedings of the First International Conference on Distributed Computing Systems (October, 1979): 651-659.
- Ensl81 Enslow, Philip H. Jr., "Distributed Data Processing --- What Is It?," AGARD Avionics Panel Symposium on "Tactical Airborne Distributed Computing and Networks," Norway, (June 22-26, 1981).
- Feld79 Feldman, J. A., "High Level Programming for Distributed Computing," Communications of the ACM 22 (June, 1979): 353-368.

- Garc79 Garcia-Molina, H., "Performance Comparison of Update Algorithms for Distributed Databases, Crash Recovery in the Centralized Locking Algorithm," Progress Report No. 7, Stanford University, 1979.
- Have78 Haverly, J. F., and Rettberg, R. D., "Inter-process Communications for a Server in UNIX," COMPCON Fall 78 (September, 1978): 312-315.
- Hoar78 Hoare, C. A. R., "Communicating Sequential Processes," Communications of the ACM 21 (August, 1978): 666-677.
- Jens78 Jensen, E. Douglas., "The Honeywell Experimental Distributed Processor - An Overview," Computer (January, 1978): 28-38.
- Morg77 Morgan, Howard L., and Levin, K. Dan, "Optimal Program and Data Locations in Computer Networks," Communications of the ACM 20 (May, 1977): 315-322.
- Ritc78 Ritchie, D. M., and Thompson, K., "The UNIX Time-Sharing System," The Bell System Technical Journal 57 (July-August, 1978): 1905-1929.
- Suns77 Sunshine, Carl, "Interprocess Communication Extensions for the UNIX Operating System: I. Design Considerations," Rand Technical Report R-2064/1-AF, June 1977.
- Thom78 Thomas, Robert H., Schantz, Richard E., and Forsdick, Harry C., "Network Operating Systems," Bolt Beranek and Newman Report No. 3796 (March, 1978).
- Zuck77 Zucker, Steven, "Interprocess Communication Extensions for the UNIX Operating System: II. Implementation," Rand Technical Report R-2064/2-AF, June, 1977.

```

<work request> ::= [ <logical net> { ; <logical net> } ]

<logical net> ::= <logical node> { <node separator>
    [ <node separator> } <logical node> }

<node separator> ::= , | <pipe connection>

<pipe connection> ::= [ <port> ] '|' [ <logical node number> ]
    [ .<port> ]

<port> ::= <integer>

<logical node number> ::= <integer> | $ | <label>

<logical node> ::= [ :<label> ] [ <simple node> |
    <compound node> ] |
    ( <simple node> | <compound node> )

<simple node> ::= { <i/o redirector> } <command name>
    [ <i/o redirector> | <argument> ]

<compound node> ::= { <i/o redirector> } '{' <logical net>
    { <net separator> <logical net> } '}'
    { <i/o redirector> }

<i/o redirector> ::= <file name> '>' [ <port> ] |
    [ <port> ] '>' <file name> |
    [ <port> ] '>>' <file name> |
    '>' [ <port> ]

<net separator> ::= ;

<command name> ::= <file name>

<label> ::= <identifier>

```

Figure 1. Work Request Syntax
(Taken from [AXIN78])

No.	Resource References		IPC		Resources Distributed on Different Nodes		Multiple Copies Resources		Some Resources on Node Other Than Home Node	
	All Visible	Some Embedded	YES	NO	YES	NO	YES	NO	YES	NO
1		X								X
2		X	X			X	X		X	
3*		X	X			X	X		X	
4*		X	X			X	X		X	
5*		X	X			X	X		X	
6		X	X		X				X	
7*		X	X		X				X	
8		X	X		X				X	
9		X		X		X			X	
10		X	X			X			X	
11*		X	X			X			X	
12*		X	X			X			X	
13*		X	X		X		X		X	
14		X	X		X		X		X	
15*		X	X		X		X		X	
16		X	X		X		X		X	
17	X			X		X			X	
18	X		X			X			X	
19*	X		X			X			X	
20*	X		X			X			X	
21*	X		X			X			X	
22	X		X			X			X	
23*	X		X		X		X		X	
24	X		X		X		X		X	
25	X		X		X		X		X	
26	X		X			X			X	
27*	X		X			X			X	
28*	X		X			X			X	
29*	X		X		X		X		X	
30	X		X		X		X		X	
31*	X		X		X		X		X	
32	X		X		X		X		X	

Figure 2. Classification of Work Requests

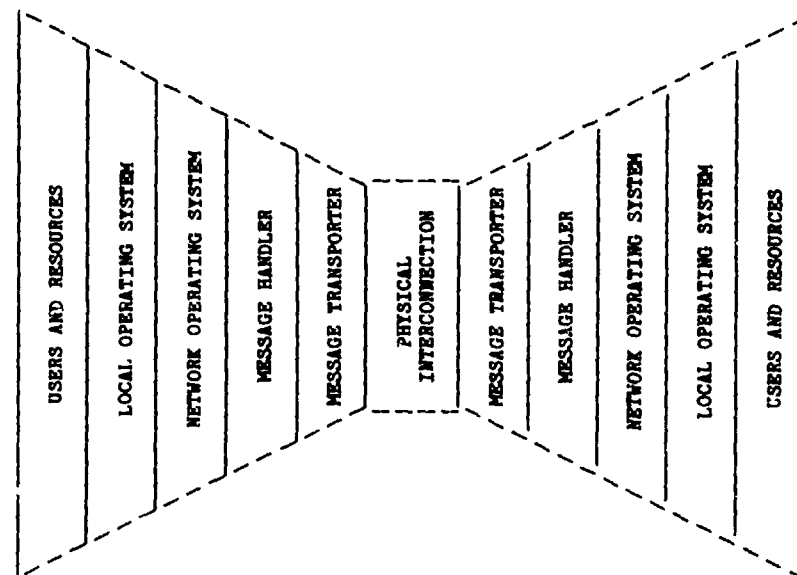


Figure 3. A Logical Model of an FDPS

EXECUTION FILE	
Name:	
Locations of candidates available:	
Location of candidate chosen:	
INPUT FILE 1	
Name:	
Locations of candidates available:	
Location of candidate chosen:	
INPUT FILE 1	
Name:	
Locations of candidates available:	
Location of candidate chosen:	
OUTPUT FILE 1	
Name:	
Locations of candidates available:	
Location of candidate chosen:	
OUTPUT FILE J	
Name:	
Locations of candidates available:	
Location of candidate chosen:	
IPC	
Input Ports:	
Output Ports:	

Figure 5. Node Control Block

Work Request:

pgm1 | pgm2 | a 2 | b : a pgm3 | pgm4 | c.1 : b pgm5 | pgm6 | 1.2 : c pgm7
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)

(0) Output port 1 of pgm1 is connected to input port 1 of pgm2.
 (1) Output port 1 of pgm2 is connected to input port 1 of the logical node labeled "a," pgm3.
 (2) Output port 2 of pgm2 is connected to input port 1 of the logical node labeled "b," pgm5.
 (3) Label for the logical node containing pgm3 as its execution module.
 (4) Output port 1 of pgm3 is connected to input port 1 of pgm4.
 (5) Output port 1 of pgm4 is connected to input port 1 of the logical node labeled "c," pgm7.
 (6) Label for the logical node containing pgm5 as its execution module.
 (7) Output port 1 of pgm5 is connected to input port 1 of pgm6.
 (8) Output port 1 of pgm6 is connected to input port 2 of pgm7.
 (9) Label for the logical node containing pgm7 as its execution module.

Data Flow Graph of the Work Request:

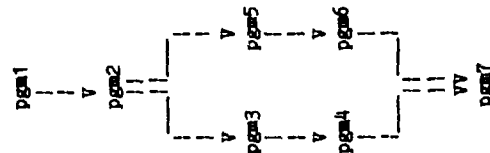


Figure 4. Example of a Work Request

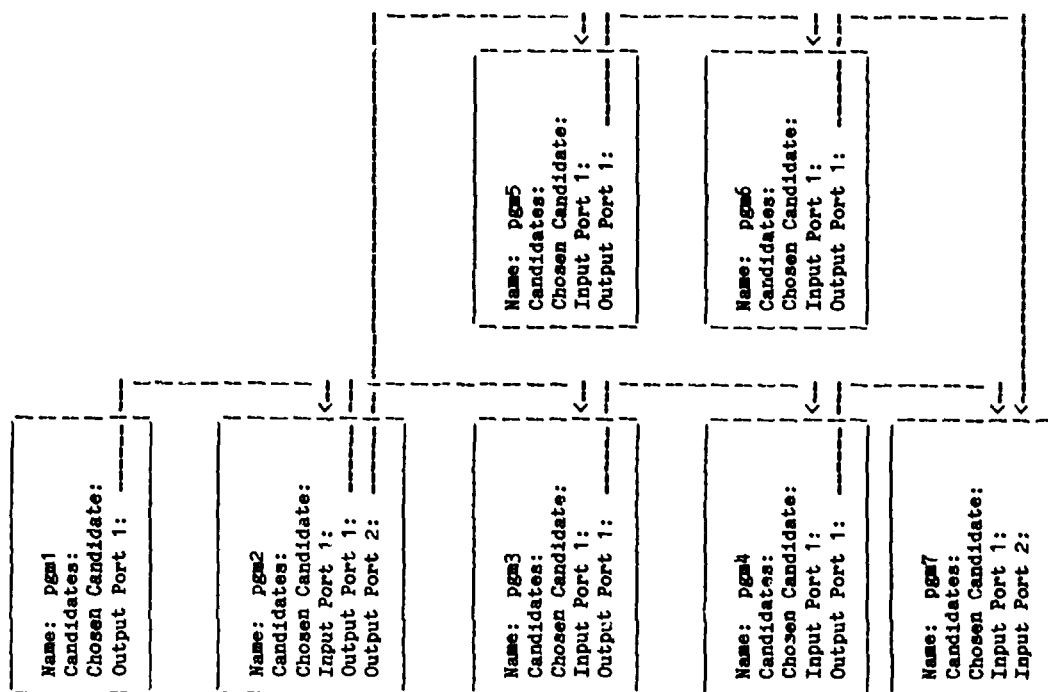


Figure 7. Example of a Task Graph Using Links within the Mode Control Blocks

(Based on the Work Request Shown in Figure 11)

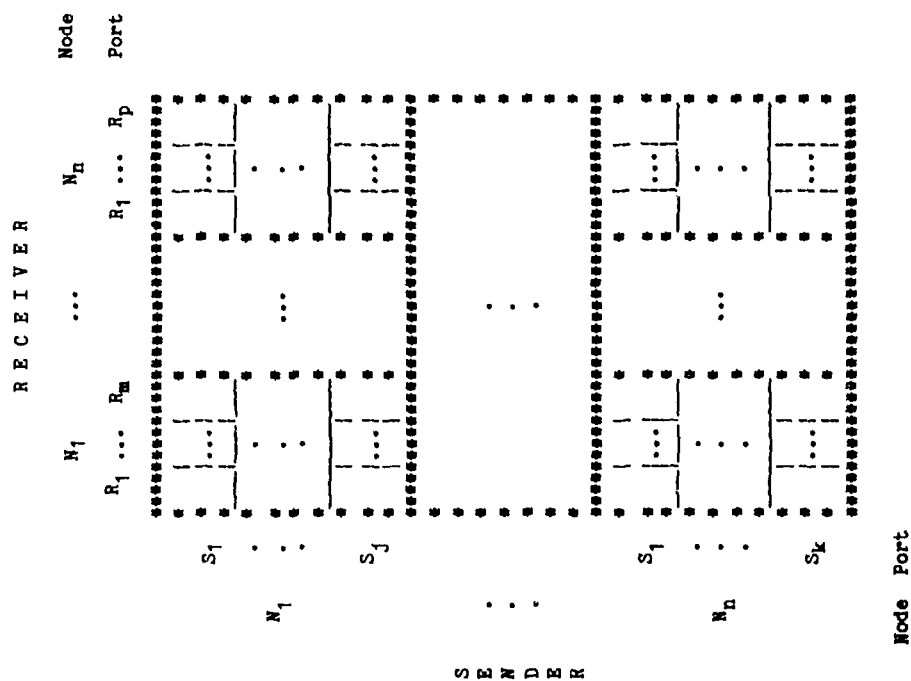


Figure 6. Node Interconnection Matrix

Figure 8. Example of a Node Interconnection Matrix
(Based on Work Request Shown in Figure 11)

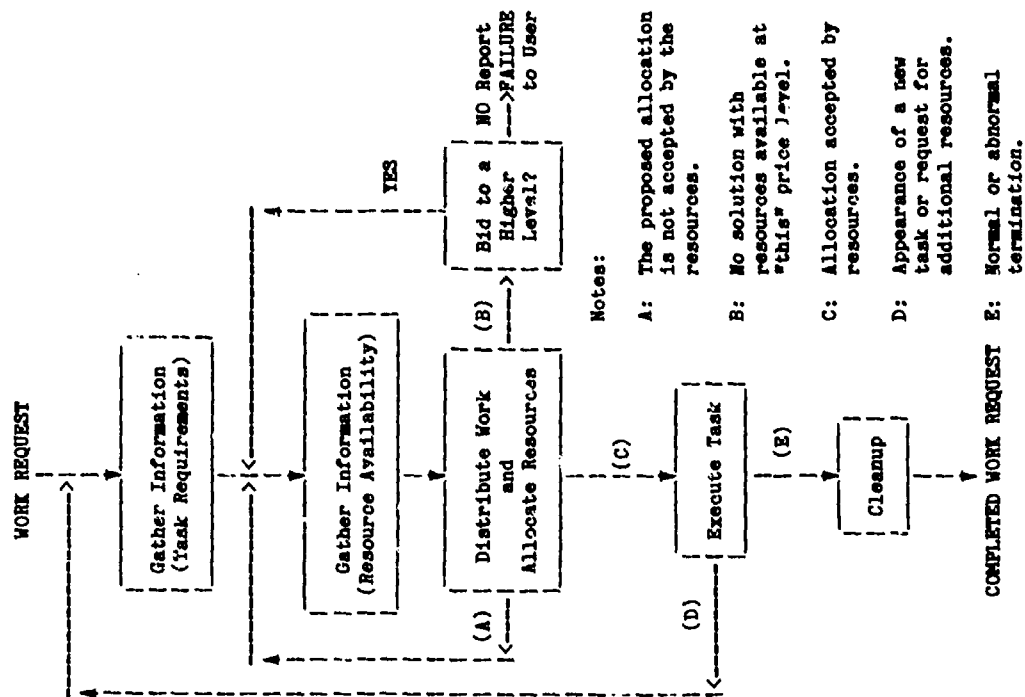


Figure 9. Work Request Processing (Detailed Steps)

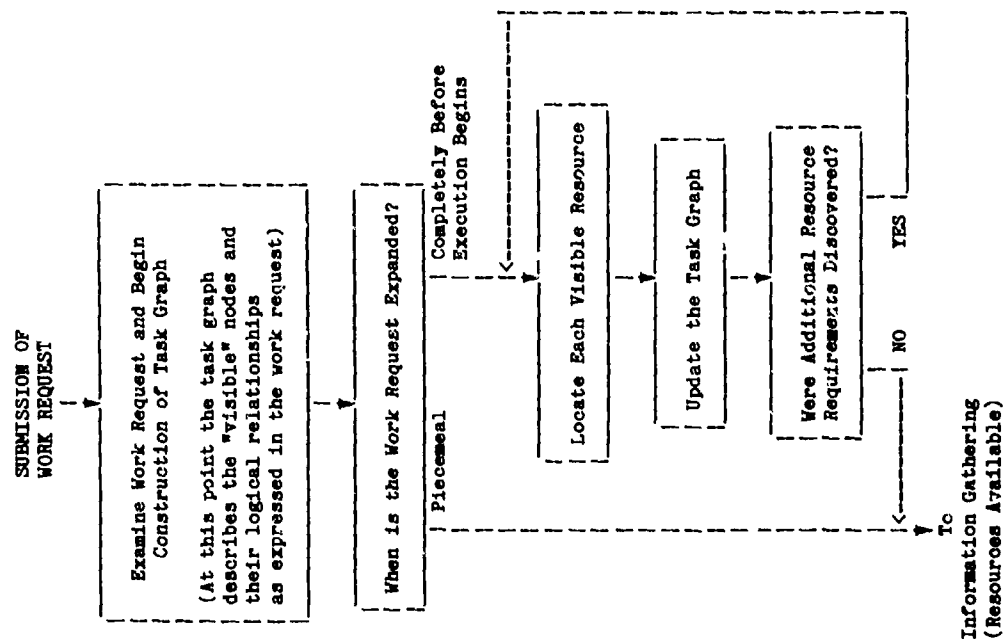


Figure 10. Information Gathering (Resources Required)

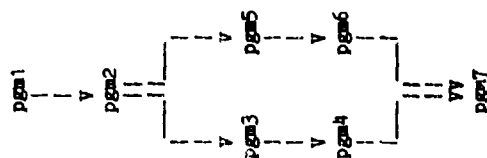


Figure 11. Example of a Work Request

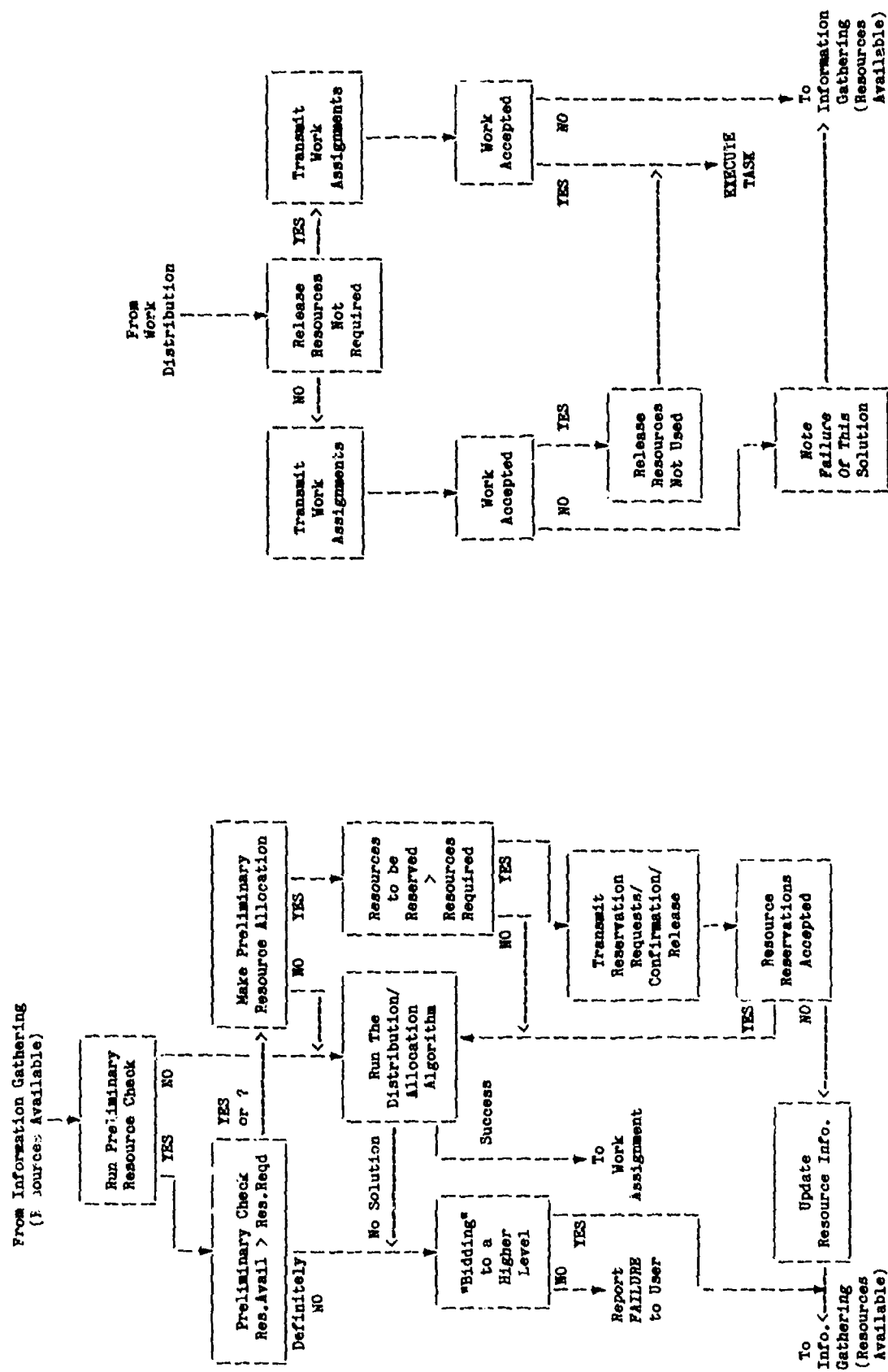
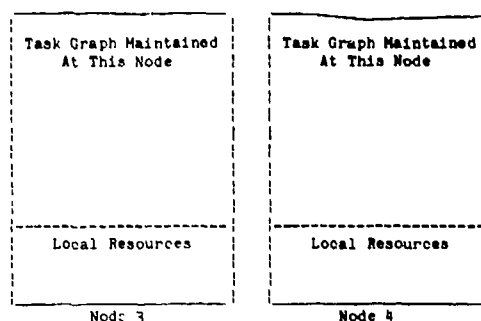
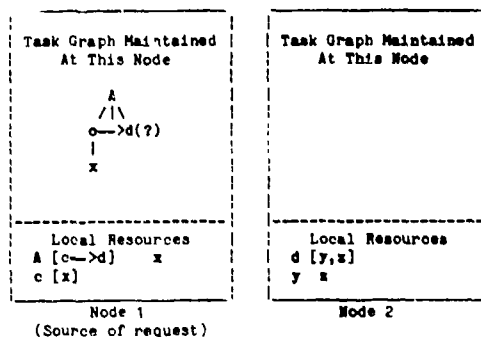


Figure 13. Work Assignment

Figure 12. Resource Allocation and Work Distribution

Request = RUN A

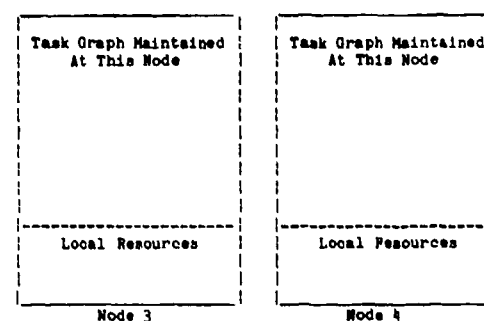
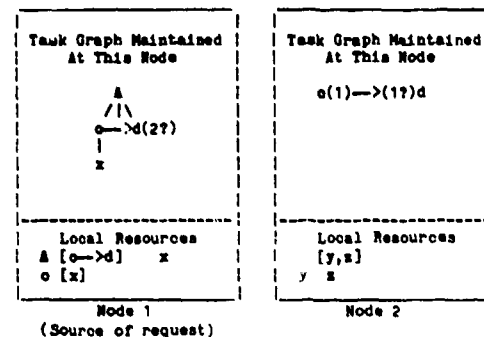
STEP 1



Comments:

- A more complex request:
- 1) Contains an explicit reference to IPC.
 - 2) Resource file located on different nodes.
- First layer is built.

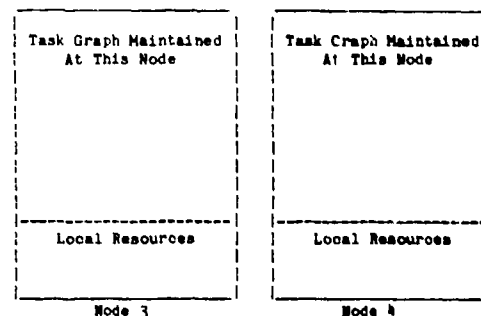
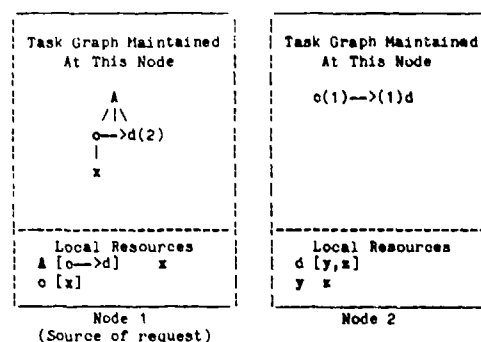
STEP 2



Comments:

File d is located on node 2 and responsibility for d is tentatively delegated to that node.

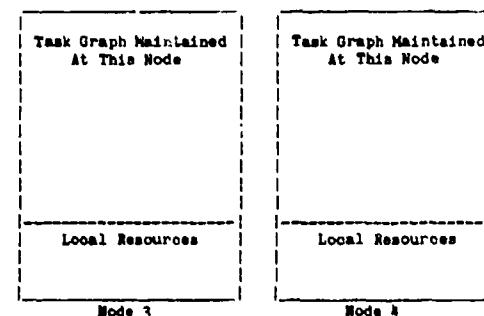
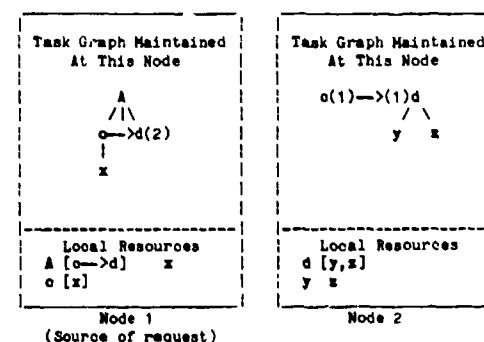
STEP 3



Comments:

Responsibility for d is accepted by node 2.

STEP 4



Comments:

The graph below d is completed.

Figure 14. Example

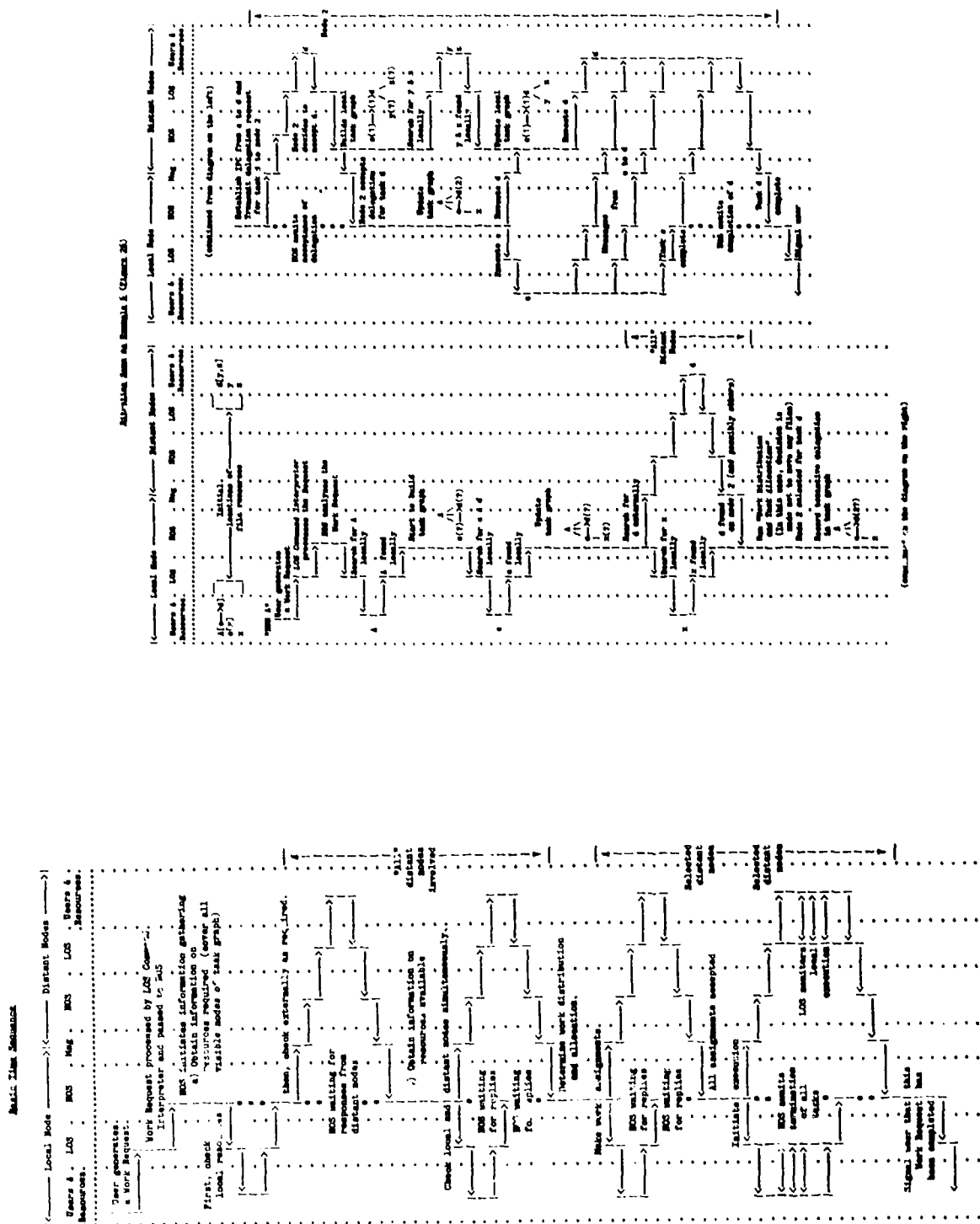


Figure 15. Basic Steps in Work Request Processing

Figure 16. An Example of Work Request Processing

RECOVERY IN DISTRIBUTED PROCESSING SYSTEMS

Liba Svobodova
INRIA
Rocquencourt
78153 Le Chesnay Cedex
France

Abstract

A powerful control abstraction called an atomic action has been developed as a general mechanism for controlling accesses to shared distributed data. In order to preserve consistency of the system, if an atomic action fails, all of its effects are undone; thus if a long complex computation is represented as an atomic action, an important amount of possibly useful work might be lost. The proposed scheme which facilitates selective internal recovery from detected errors, node failures, and communication failures employs nested atomic actions. When an atomic action terminates, its results are not made permanent until the outermost atomic action is committed, but they survive local node failures. Each subtree of nested atomic actions is recoverable (undoable) individually, thus making it possible to switch to an alternative algorithm, service, or physical node upon a failure. Finally, a recovery point is established in stable storage as part of a remote request, so that work done outside of the requesting node is not lost if this node fails.

1. INTRODUCTION

A distributed system, as viewed in this paper, is a network of computing nodes which, although they have to cooperate in some predetermined manner, maintain a fair degree of autonomy with respect to their internal organization and management [CLAR 80, SVOB 79A, SVOB 79B]. The communication subsystem facilitates exchange of messages between any two nodes, but does not guarantee it at all times. Individual nodes provide certain services to the rest of the system. These services are not memoryless: while they can be provided only if adequate hardware resources are available at the nodes, they contain another critical component, and that is stored data.

Distributed systems are often claimed to be inherently more reliable than systems that are built on the top of a single central processor. First, propagation of low level errors is restricted by physical separation of processes and resources. Second, if one node fails, it might be possible to finish the computing tasks in progress by using services of another node. However, distributed systems introduce also new reliability problems, the most basic one being the difficulty of maintaining a globally consistent state of the system. Given that the programs of the individual tasks are correct, the problem of maintaining a consistent state becomes a problem of synchronization and recovery. In a distributed system, the difficulty of recovery is in part due, paradoxically, to the fact that a failure of a single node does not disable the whole system. The other important aspect is the uncertainty brought about by the imperfect communication subsystem: from the point of view of a node requesting a service, a failure of the communication subsystem to deliver the request or the response is, in general, indistinguishable from a failure (crash) of the node providing the service.

The problem of recovery in a distributed system has been studied mostly in the context of database management. A logical unit of work is represented by a transaction [TRAI 79, GRAY 80]. Transactions are assumed to preserve certain application specific integrity constraints defined on the data, as well as the integrity constraints of the data structures representing the database. To maintain the integrity constraints, if an error is encountered during execution of a transaction, the transaction is aborted and all of its effects are undone. A related aspect is that of the stability of results: if a transaction completes, its effects are guaranteed to be permanent, that is, the results will not be lost or damaged by subsequent system failures; this is again a recovery problem, although at a different (lower) system implementation level.

A computation in a distributed system may not be able to proceed normally for many different reasons:

- the invoker decided to abort it
- the inputs were incorrect
- an unrecoverable hardware or software error was encountered during execution
- a scheduling conflict was encountered
- one of the involved nodes failed
- communication failed.

As said above, in the simple transaction model used in distributed database management systems, all of these situations are treated in the same way: the transaction is aborted and its effects are undone. For long, complex computations, a lot of work might be wasted if this policy is followed. Thus, in addition to guaranteeing data integrity and stability, an important goal is to complete computations in spite of errors and failures of different system components. In particular, since the same of similar service might be provided by several nodes, a failure of some node or a failure to communicate with a particular node does not have to abort all computations requiring such a service. Also, if a failed node can recover in such a way that it does remember the state of the computations that were running on it at the time of the crash, these computations can be completed without having to seek alternative resources or alternative solutions. This paper focuses on this problem of resiliency, and in particular, resiliency with respect to node and communication failures.

2. GENERALIZED MODEL OF DISTRIBUTED COMPUTATIONS

The transaction model assumed in most studies of recovery issues in distributed database management systems is limiting from yet another point of view: in general, database transactions are assumed to have a very flat (usually just one level) strictly hierarchical structure. A transaction has a coordinator and several data managers (workers, agents) that manage different parts of the database, but there are no lower level dependencies between these data managers. More general distributed computations might present the sort of problem depicted in Figure 1. In this example, the top level program is initiated at node A. This program includes requests for service S_i from node B, and service S_j from node C. A short notation $S_m.N$ will be used throughout this paper where S_m specifies the service requested and N the name of the node providing the service. The services provided by the individual nodes might be much more complex than "read data" and "write data" usually assumed to be the only types of requests in database transactions. Following the concepts of structured programming, the actual implementation of these services is unknown to the invoker. Thus the program at node A does not know that requests $S_i.B$ and $S_j.C$ both, as part of their implementation request services from node E and that each such request results in an update of a data object X .^{*}

If the programs that implement the services $S_i.B$ and $S_j.C$ are executed concurrently, their proper synchronization during normal execution represents practically the same problem as the problem of the synchronizing database accesses of independent concurrent computations. The problem that will be studied here is the possibility of recovery of the individual requests. Assume that the request sent to node B fails, but by that time node C has already done a significant amount of work as a result of the request received from node A. As a response to a failure of the request $S_i.B$, the requesting program at node A might try one of the following alternatives:

1. retry request $S_i.B$
2. search another node that provides the same service as node B
3. try an alternative algorithm (different service) that produces possibly different kinds of results, but still satisfactory (less accurate, for example).

At the programming level, such alternatives could be specified with the aid of a construct called a recovery block [RAND 75]. However, before an alternative can be tried at any level, it is necessary to restore the state of the resources used by the failed branch of the computation. If object X has already been modified as a result of the failed request $S_i.B$, and if this modification has been seen by the other branch that originated at node C, it might be necessary to undo indeed everything. The main point is, however, that these dependencies are not known at the level of node A: unless some control mechanisms are added, it is always necessary to account for the worst case, and to undo everything. It should be noted that this kind of problem will be encountered even in a single processor system, if the "nodes" are just separate modules such as, for example, the guardians [LISK 79, SVOB 79A]. However, additional problems occur in a network of physical nodes, as will be seen later.

3. ATOMIC ACTIONS

A general mechanism for solving the problem of consistency in the presence of concurrent computations and asynchronous faults is a construct or a control abstraction called **atomic action**. From the point of view of the invoker, an atomic action is an operation the effects of which are determined entirely by its algorithm. Atomic actions are:

1. indivisible with respect to concurrent computations: the intermediate results of one atomic action cannot be modified or observed by concurrent computations;
2. indivisible with respect to failures: an atomic action either terminates normally and produces a new consistent state as defined by its algorithm, or has no effects.

In transaction-oriented database management systems, the transactions are in fact atomic actions; however, the concept of an atomic action is more general than that of an update of a shared database.

From the implementation point of view, an atomic action can be viewed as a control sphere that encompasses a set of resources, both shared and private. An atomic action can be executed by a single process, if all these resources are in the same physical node, or it might involve several processes. The resources could be all acquired at the beginning of the execution of the atomic action, however, often this is not possible since the complete set of the required resources is not known at that time. For example, the "resources" might be records of a database. Which records will be read or modified might depend on the value of certain fields of some other records. One solution is to "acquire" the whole database. A more effective solution is to let the atomic action acquire needed records during the course of execution, as the need is determined. This necessitates synchronization protocols that properly order the elementary execution steps of different atomic actions, and resolve scheduling anomalies. Basically, it is necessary to ensure that a set of atomic actions executed concurrently is serializable [ESWA 76]. If an atomic action fails, the resources that it has acquired have to be restored (recovered) to their state at the time of their acquisition, and released; to ensure that no other computations have been affected by such a failure, the resources are not released until the atomic action terminates.

Many sophisticated mechanisms have been proposed to provide atomicity in distributed systems. Serializability of atomic actions is achieved either by locking protocols or by a priori ordering of requests belonging to different atomic actions by associating with them globally unique timestamps. In this paper, only the mechanisms needed to assure atomicity from the point of view of failures will be discussed. Also, while it would be interesting to consider different types of resources, the resources of an atomic action are assumed to be data objects. The key problems then are: 1. coordination of the changes to the physical representation of

^{*} A more general transaction model that covers situations of this kind is developed in [LIND 79]. However, the emphasis in this model is on detecting node crashes, after which the whole transaction is aborted. Also, a transaction can be executing only on a single node at a time.

objects updated within the same atomic action, ii. their commitment, that is, making these changes permanent and visible to other computations, iii. object recovery, that is, restoration of an object to its previous state, and iv. coordination of the recovery of the objects modified by a failed atomic action. These are non-trivial problems even if all objects are stored at the same node and the atomic action involves only a single process; in a distributed system, the inherent uncertainty and the cost of internode communication add another dimension to this problem.

In order to be able to execute arbitrary computations as atomic actions, it is necessary that the elementary steps of which these computations are constructed are also atomic. In particular, physical updates of data on storage devices must be atomic. In general, to guarantee that stored data will survive node crashes, they must be stored on non-volatile secondary storage devices, since the usual recovery from a crash is to reinitialize the system, which means that from the point of view of normal access, the previous content of the primary memory is effectively lost.^{*} But such storage is not yet stable; additional procedures and mechanisms (e.g. duplication, checkpoints + log) are needed in order that stored information survives device crashes and spontaneous decays. However, the system could crash during a write operation, when part of the data has already been overwritten with a new value; this would leave the data object in an undefined state. Stable storage that guarantees that a write operation is either performed correctly or has no effects is called atomic stable storage. Efficient implementation of a storage system with such properties is still a research issue [LAMP 79, SVOB 80]; in this paper, it is assumed that all nodes provide stable storage and that information stored there can be changed atomically, although it does not necessarily mean that such information is updated in place.

4. NESTED ATOMIC ACTIONS

From the recovery point of view, atomic actions can be viewed as a damage confinement mechanism: while it is generally assumed that everything within the failed atomic action is suspect, the mutual exclusion mechanisms of atomic actions guarantee that nothing outside has been affected. The damage confinement is a very useful property since it makes computations that are implemented as atomic actions separately recoverable. However, as already argued, the assumptions about the damage within an atomic action is often unnecessarily strict.

An alternative to aborting the entire atomic action is to set up recovery lines within it: when an error is detected, the computation has to be backed out only to the nearest recovery line. If an atomic action involves just a single process, a recovery line consists of a single recovery point (checkpoint) that contains the state of that process. If several processes are involved, then recovery lines can be either prearranged, or determined dynamically. The beginning of an atomic action represents a preplanned recovery line. However if processes do not set up recovery lines in a coordinated manner, where the nearest recovery line is at the time when an error is detected is not obvious. Merlin and Randell developed "chase protocols" for determining recovery lines dynamically [MERL 77]. This work was extended by Wood who worked out a protocol for keeping track of the dependencies between processes (propagation of information) and for determining when it is safe to discard a particular recovery point [WOOD 81]. The approach taken here is essentially to preplan the recovery structure, and to tie it to the logical structure of the program.^{**}

The basic solution is to use nested atomic actions: each atomic action can be built of smaller atomic actions that can be executed either sequentially or in parallel, and that will be properly synchronized with respect to use of shared data objects. Reed developed an integrated set of mechanisms for implementation and control of nested atomic actions [REED 78]; these mechanisms will be extended here to facilitate selective internal recovery.

In Reed's model, each atomic action is represented by two entities: a pseudo-temporal environment and a commit record. The pseudo-temporal environment is the mechanism that assures serializability of atomic actions. The commit record is a data structure that contains the state of the atomic action. The commit record is created with the state set to "undefined". When the atomic action terminates normally, the state is set to "committed", otherwise, if the termination is abnormal, the state in the commit record is set to "aborted". An atomic action a_{ix} which is nested within an atomic action a_i is made dependent on the outcome of a_i : this dependence is recorded in the commit record of a_{ix} in the form of a reference to the commit record of a_i . Commit records are stored in atomic stable storage. Finally, all requests to create, read, update, or delete an object include a reference to the commit record of the atomic action within which the request is made.

When an object is updated, the system creates a new stable version of this object without destroying the old one. This version contains a reference to the commit record of the atomic action under which it was created. As long as that commit record is in the state "undefined", only the atomic action that created that version can read it. Once this atomic action terminates, its commit record is set to the state "committed", but it does not mean that the new version can be used freely from anywhere within the system: its fate still depends on the outcome of the enclosing atomic actions. However, once committed locally, a new version can be used from anywhere within the invocation subtree rooted by the nearest enclosing atomic action that is still in the state "undefined", since if this atomic action is eventually aborted, all of its dependents will be aborted anyway. When an atomic action is aborted, all of the object versions created by it and by all of its dependents are discarded, but this does not affect other branches of the invocation tree, since they could not have seen the invalidated versions. Once the top level atomic action reaches the final state, be it "aborted" or "committed", this information is propagated to its dependents and successively to their dependents and encached in their commit records.

^{*}) It is quite difficult to find a simple definition of "system crash"; in this paper, it will be assumed that a crash is any event that causes such complete reinitialization.

^{**}) A similar approach is used by Shrivastava, but he assumes that recovery might be provided on a more abstract level, under the direction of a manager of an abstract type [SHRI 81].

Let us return to the example given in Section 2. The main program at node A will be, of course, an atomic action, but in addition each remote request will start a new atomic action in the receiving node. It is assumed that each request returns a response when the atomic action created by that request terminates. It is the responsibility of the requestor to wait for the response before its atomic action is committed.

Now assume that the request $S_{j,E}$ from node B is the first one to arrive at node E; this situation is depicted in Figure 2. Once the execution of this request is finished, it is possible to process the request $S_{k,E}$ from node D, but not $S_{i,E}$ from node C, since the atomic action rooted at node B has not finished. Figure 3 shows a situation when $S_{i,B}$ terminated normally and a new version of object X has been created finally by the request $S_{i,E}$ from node C. If the request $S_{i,B}$ failed for some reason, both versions X_1 and X_2 would be discarded before $S_{i,E}$ could proceed. Of course, it is assumed that there do not exist any precedence constraints between the updates performed on X, otherwise the requests $S_{i,B}$ and $S_{j,C}$ could not be executed concurrently, without any explicit synchronization on their level.

5. CRASH RECOVERY

The mechanisms described in the preceding section are sufficient for orderly recovery from errors that are either reported or can be safely detected by the invoker of a request for service. In the given example, it would mean that if the request $S_{i,B}$ fails, either node B sends an error message to node A or A detects an erroneous response. In either case, A receives some response from B. As said earlier, object versions and commit records are stored in stable storage, thus they survive node crashes. This means that if, for example, node E crashes after it has sent back a response to the request $S_{k,E}$, this crash can have no effect on the results of that particular call. However, if an invoker does not receive a response to its request, the situation becomes more complicated. Namely, to prevent that a node waits indefinitely for a response from another node, it is necessary to set a timeout for each remote request. However, when the timeout expires, it is not possible to deduce the state of the atomic action created by that request. Any of the following might have happened:

1. the target node Z never received the request (the communication subsystem did not deliver the message)
2. the request was executed but terminated abnormally
3. execution of the request was interrupted by a crash of node Z
4. execution of the request terminated normally, but the response was not delivered to the requestor (either the node Z crashed before the response could be sent, or the communication subsystem failed to deliver the message)
5. execution of the request still continues (either the timeout was set too short or the execution is slower due to high load or the need to recover from internal errors).

When the timeout expires, the invoker may decide either to repeat the request or try an alternative service, or an alternative algorithm. Let us postpone the discussion of the first possibility until the next section and analyze the problem of switching to an alternative. For the first two situations listed above nothing special has to be done since the failed request had no effects. In the other three cases, the atomic action started by the request either has been or might be locally committed (in case # 3, this assumes that the node recovers in such a way that it is capable of resuming the computations interrupted by the crash). Its commit record contains a reference to the commit record of the directly enclosing atomic action, that is, the atomic action of its invoker; later, when the state of the commit record of the invoker is set to "committed", the whole subtree abandoned when the invoker switched into an alternative algorithm would be in fact committed! Thus it is necessary in some way to invalidate the reference in the commit record of a dependent atomic action declared to have failed on the basis of a timeout. The commit record of an atomic action should reside on the same node as the objects manipulated by the atomic action, that is, in the given model on the node on which the atomic action is executed. This means, however, that if no response is received from this node, it must be assumed that the commit record, if it exists, is also inaccessible and therefore the reference to the commit record of the invoker cannot be removed. A possible solution shown in Figure 4 is to add to each commit record a list of the identifiers of the current dependent atomic actions. In addition, each atomic action will contain its own id in its commit record. The identifier of a dependent atomic action is generated by the invoker (although it could be generated by some third party) and included in the request sent to the node providing the service. When the invoker decides that a particular request has failed, it removes its id (that is, the id of the atomic action that might have been started by the request) from the list in its own commit record. Before the results of a dependent atomic action a_{ix} can be committed up to the level of its invoker a_i , it is necessary to check if the identifier of a_{ix} is still on the list in the commit record of a_i .

If a node is to resume local computations interrupted by a crash, and this is important in particular when a computation had made remote requests, it is necessary for each such computation, to remember not only its local state, but also its interactions with other nodes. Some of this information is already in the commit record, however, it is also necessary to remember the outstanding requests. Thus a checkpoint should be established in stable storage as part of a remote request. A remote request thus should include the following steps:

- i. the invoker generates a new identifier ID for the dependent atomic action
- ii. this identifier is included in the list in the commit record of the invoker; the commit record is updated atomically in stable storage
- iii. a checkpoint is made which includes a reference to the commit record and the message to be sent
- iv. the message which includes the identifier ID and a reference to the commit record of the invoker *) is sent to the target node
- v. on failure: remove ID from the list in the commit record of the invoker; the commit record is updated atomically in stable storage

*) A reference to a commit record could be an identifier of the actual object that represents the commit record or the identifier of the atomic action represented by that commit record.

If the node crashes after the checkpoint but before another checkpoint is established, the request will be resent, thus the target node must be able to detect when a received request is a duplicate. Although many communication subsystems detect and suppress duplicate messages, their mechanisms are not sufficient, since from the point of view of the communication subsystem, each retry represents a different message. However, if the request has been previously received, then the receiving node must contain a commit record with that ID; detection of a duplicate is therefore simple. Finally, if the invoking node crashes during step v but before the ID has been removed from the list, again the request will be resent; at this time, it might actually succeed, if the "failure" detected previously was a result of a timeout, but this does not cause any inconsistency.

6. PROGRAMMING ASPECTS

As already mentioned in Section 2, a programming construct called a recovery block can be used to specify the alternatives to be tried in case that a particular request fails. The structuring imposed by recovery blocks also provides another solution to the problem of branches abandoned as a result of a timeout discussed in the preceding section. Figure 5 shows a possible structure of the program running in node A that uses recovery blocks. A remote procedure call is used as a means for making remote requests. Since when such a call is made, the calling process must wait for a response, in order to be able to process requests S_i.B and S_j.C in parallel, it is necessary to make the respective calls in different processes in node A. In the given example, this is indicated by the enclosing parbegin/parsnd structure, although processes could be forked in a more general manner, for example, just before a remote call is made. It is assumed that a timeout is associated with each remote call; if the timeout expires, the call terminates by signalling an exception. This exception and any other abnormal return, if not handled within the enclosing block, will result in a switch into an alternative program within the same recovery block. If all alternatives fail, failure is signalled to the next enclosing block, which, in this case, is the topmost level. Since no alternative is specified at this level, the whole computation would be aborted.

According to the semantics of recovery blocks, before an alternative can be tried at any level, it is necessary to return to the initial state of the recovery block, that is, undo what has been done by the failed alternative. Considering that it is also necessary to coordinate accesses to shared resources from different recovery blocks executed in different processes, each alternative of a recovery block should be, in fact, a separate atomic action. Figure 6 shows the new tree of commit records for the same execution state as the one that was depicted in Figure 2: additional commit records were added for the recovery blocks that enclose the individual remote calls.

When a remote call fails, then in order to abandon that particular branch, the alternative from which the call was made is abandoned also, and its commit record is set to "aborted". When another alternative is tried, a new commit record is created for it. Thus even though the remote request might be finished later (in case that the remote call failed because of a timeout, after possibly several retries), its results can never become erroneously committed. This means that it is not necessary to keep the list of current dependent atomic actions in the commit record, as proposed in the preceding section. Or, viewed differently, this list now consists of the commit records of the current alternatives.

Let us return now to the question of what has to be done if, after a timeout, the remote call is retried. It might seem that this is the same problem as if the request was resent as part of recovery from a crash, but the situation here is a little bit more complicated. At this level, whether or not to retry a request is the decision of the programmer. If it is the programmer who in order to send a request to another node has to write the individual steps of the program P₁ outlined in the preceding section, then the request can be resent in the following way:

```
P2 :  o.  set retry := n
      i.  get new ID
      ii. create a checkpoint which includes a reference to the commit record*) of the invoker and
           the message to be sent.
      iii. send the message which includes ID and a reference to the commit record of the invoker
      iv. on timeout : retry := retry - 1
           if retry ≥ 0, repeat step iii
           else failure
```

The request will be retried up to n times, each time with the same ID; thus this is indeed the same problem as if the request is retried after a crash. On the other hand, the programmer could be given a primitive "remote-call" which consists of the steps i to iii of P2. In order to retry a request, it is necessary to repeat the call:

```
P3 :  0.  set retry := n
      1.  remote-call (service, node, parameters)
      2.  on timeout : retry := retry - 1
           if retry ≥ 0, repeat step 1
```

Each time the remote-call is repeated, a new ID is generated; thus the receiving node, a repeated call looks like a new request. This means that the effects of the previous try, if the request was indeed received and executed, must be undone. Thus, in connection with recovery blocks, the whole alternative of the recovery block that contains the call should be repeated. A more graceful solution is to provide a remote-call primitive that includes the option of an automatic retry, that is, in its implementation it includes the steps o and iv of P2. Thus the language should provide a primitive remote-call (service, node, parameters, n) where n is

*) The commit record still must be included in the checkpoint, since it is part of the state of a computation.

the number of retries desired.*)

Many arguments have been raised recently with respect to the basic communication primitives for a distributed system, the primary aspect being the choice between remote procedure calls and more general send and receive primitives [LISK 79, LAUR 79]. Although in order to achieve desired concurrency a separate process has to be forked for a remote call, this combination seems to provide a cleaner structure, particularly from the point of view of recoverability. The same effect could be of course achieved with two separate send and receive primitives, but if the send and receive parts of different requests are interleaved, it will be more difficult to determine the proper recovery structure. It should be noted that in the context of the recovery model presented here, a remote node must reply to the requestor even if no data is sent in the reply; thus having a simpler send primitive that does not wait for a response does not provide any advantage. However, both the recovery model and the communication primitives require further study.

CONCLUSION

The concept of an atomic action as a general mechanism for controlling recovery in computer systems and particularly in distributed systems is gaining more and more acceptance. Of course, there is always the problem of cost. The heavy use of stable storage and the extra messages needed to test dependencies of nested atomic actions and to coordinate their commitment or abortion can be very expensive. However, if a very reliable system is needed, alternative mechanisms might be equally expensive. Atomic actions have some strong advantages. They provide a uniform scheme for coping with either local or remote failures. Nested atomic actions support naturally common programming techniques. What is needed is more of experimental work that uses these concepts to demonstrate that it is indeed feasible to build in this way not just a very reliable but also a practical system.

REFERENCES

- CLAR 80 Clark, D.D., Svobodova, L., "Design of Distributed Systems Supporting Local Autonomy", Digest of Papers, COMPCON Spring '80, San Francisco, California, February 1980, pp. 438-444.
- ESWA 76 Eswaran, K., et al., "The Notions of Consistency and Predicate Locks in a Database System", Comm. of ACM, Vol. 19, N° 11 (November 1976), pp. 624-633.
- GRAY 80 Gray, J., "A Transaction Model", Lecture Notes in Computer Science, Springer-Verlag, Vol. 85, July 1980, pp. 282-298.
- LAMP 79 Lampson, B., Sturgis, H., "Crash Recovery in a Distributed Data Storage System", XEROX PARC, Palo Alto, California, 1979 (to appear in Comm. of ACM).
- LAUR 78 Lauer, H.C., Needham, R.M., "On Duality of Operating System Structures", Proc. of Second International Symposium on Operating Systems, IPIA, Rocquencourt, France, October 1978.
- LIND 79 Lindsay, B.G. et al., "Notes on Distributed Database", IBM Research Laboratory Technical Report N° RJ2571, San Jose, California, July 1979.
- LISK 79 Liskov, B., "Primitives for Distributed Computing", Proc. of 7th ACM Symposium on Operating Systems Principles, December 1979, pp. 33-42.
- MERL 77 Merlin, P.M., Randell, B., "Consistent state Restoration in Distributed Systems", Technical Report N° 113, University of Newcastle upon Tyne, Newcastle upon Tyne, England, October 1977.
- RAND 75 Randell, B., "System Structure for Software Fault Tolerance", IEEE Transactions on Software Engineering, Vol. SE-1, N° 2 (June 1975), pp. 220-232.
- REED 78 Reed, D.P., "Naming and Synchronization in a Decentralized Computer System", MIT Laboratory for Computer Science Technical Report 205, Cambridge, Massachusetts, September 1978.
- SHRI 81 Shrivastava, S.K., "Structured Distributed Systems for Recoverability and Crash Resistance", IEEE Transactions on Software Engineering, July 1981 (to appear).
- SVOB 79A Svobodova, L., Liskov, B., Clark, D., "Distributed Computer Systems : Structure and Semantics", MIT Laboratory for Computer Science, Technical Report N° TR-215, Cambridge, Massachusetts, March 1979.
- SVOB 79B Svobodova, L., "Reliability Issues in Distributed Information Processing Systems", Proc. of the Ninth IEEE Fault tolerant Computing Symposium, June 1979, pp. 9-16.
- SVOB 80 Svobodova, L., "Management of Object Histories in the SWALLOW Repository", MIT Laboratory for Computer Science Technical Report 243, Cambridge, Massachusetts, July 1980.
- TRAI 79 Traiger, I.L., et al., "Transactions and Consistency in Distributed Database Systems", IBM Research Laboratory Technical Report RJ 2555, San Jose, California, June 1979.
- WOOD 80 Wood, W.G., "Recovery Control of Communicating Processes in a Distributed System", University of Newcastle upon Tyne, Technical Report N° 158, Newcastle upon Tyne, G.B., November 1980.

*) Note that if it is indeed desired to start a new atomic action on a retry, it is still possible to use the sequence P3, where the last parameter in the remote-call is set to 0.

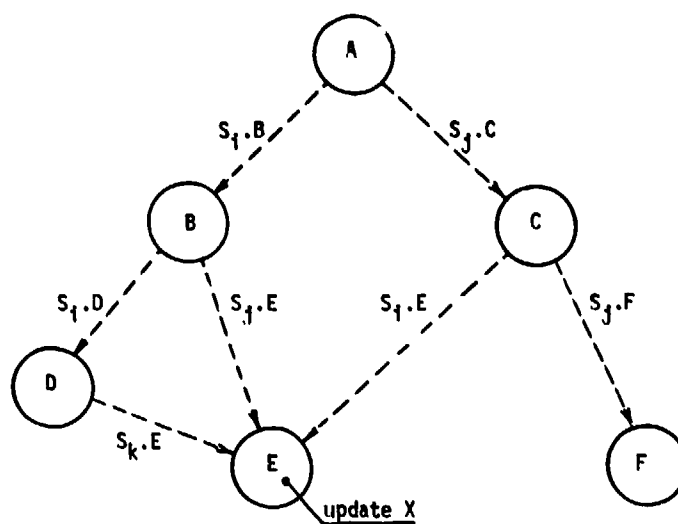


Figure 1: Example of a distributed computation with internal sharing

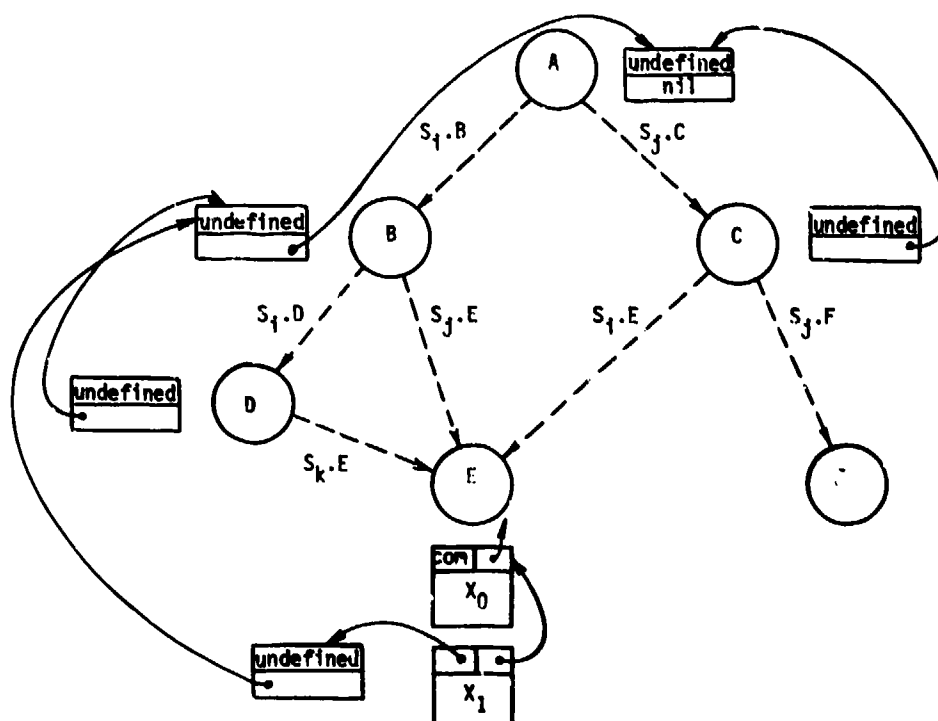


Figure 2: State of the object X and of the commit records of the enclosing atomic actions before the termination of the request $S_j.E$

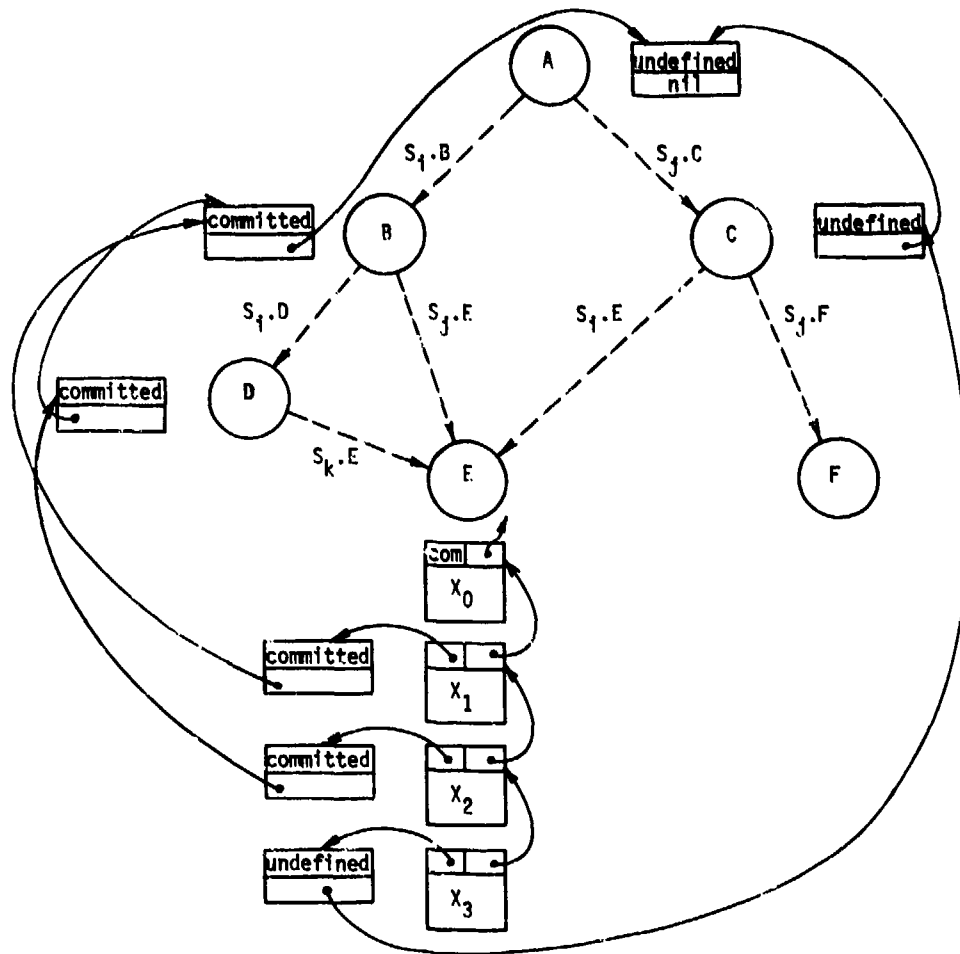


Figure 3: Situation during the execution of the request $S_i.E$

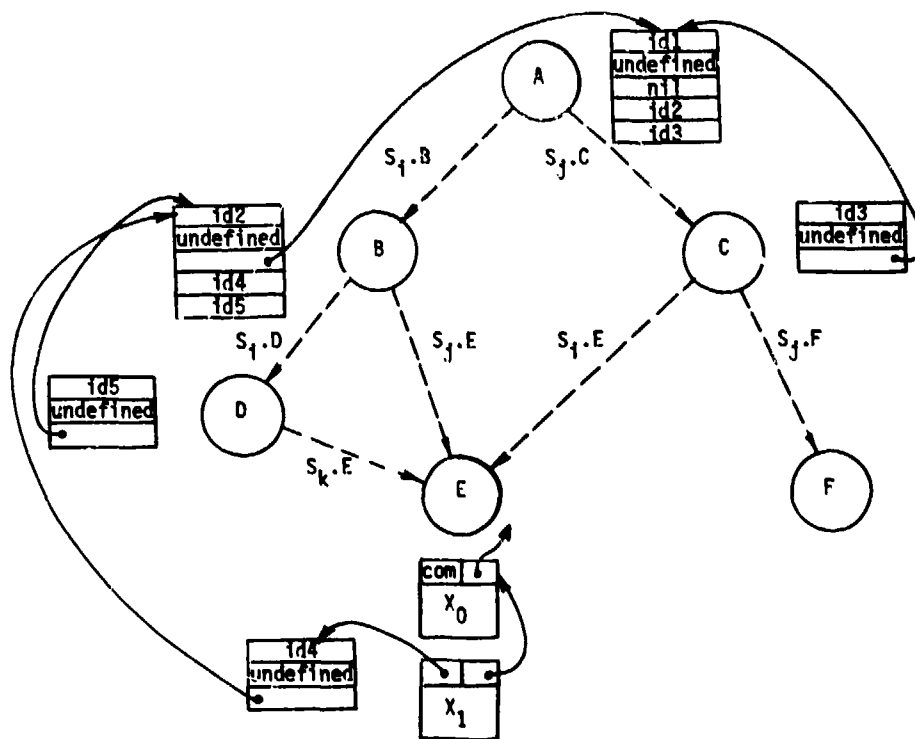


Figure 4: Same state of execution as in Figure 2; identifiers of atomic actions were added for crash recovery

```

A: ensure A.test
  by AO: parbegin
    AB: ensure AB.test
      by AB1: begin
        :
        : remote_call( $S_i, B, parameters$ );
        :
        : end
      else by AB2: begin
        :
        : remote_call( $S_i, B', parameters$ );
        :
        : end
      else error
    AC: ensure AC.test
      by AC1: begin
        :
        : remote_call( $S_j, C, parameters$ );
        :
        : end
      else by ...
      else error
    parend
  else error

```

Figure 5: Structure of the program executed at node A that uses recovery blocks

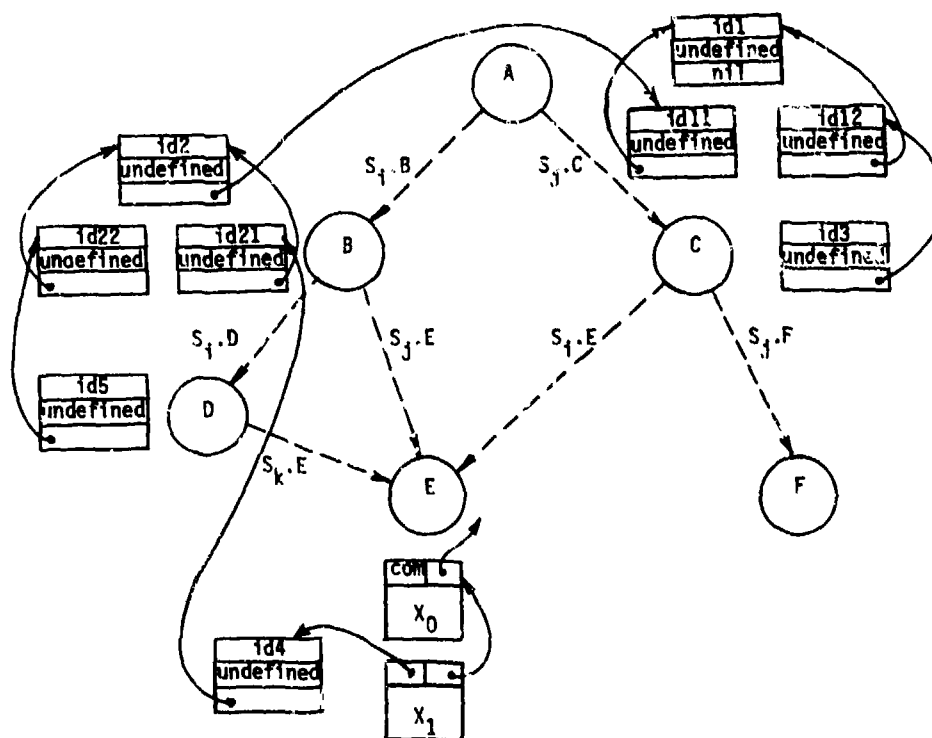


Figure 6: Same state of execution as in Figure 2; each remote request is made from a separate recovery block

GENERALIZED POLLING ALGORITHMS FOR DISTRIBUTED SYSTEMS

Jack Keil Wolf
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, Massachusetts 01003 USA

ABSTRACT

A polling algorithm for a distributed system is an algorithm which can be simultaneously run at all terminals in a network and which has as its aim the determination of which terminals have a positive response to a specific query. Of particular interest is the situation where one expects very few of the terminals to respond positively and where a terminal signifies a negative response by not transmitting at all. In such a case it is inefficient to poll the terminals in a round-robin manner. A more efficient procedure is to group the terminals into subsets in which all terminals in a subset are queried simultaneously. Then if all respond negatively no further queries need be addressed to that subset. If the responses from the terminals in the subset are mixed then this subset is further subdivided into smaller subsets until the responses of all the terminals are determined.

In this paper two distinct algorithms for polling are considered. In both algorithms, the terminals of the network are represented by leaves in a binary tree and the subsets are subtrees in the overall tree. The two systems differ in the assumptions made regarding the types of responses sent and how the responses are interpreted. The performance of these two schemes are compared with each other and with ordinary round-robin polling.

1. INTRODUCTION

Consider a set of communication terminals (or nodes) which communicate over a common communication channel and for which every terminal can reliably receive the transmission of every other terminal. Suppose that a query is to be made of all terminals in the network and that it is desirable for every terminal to know the yes/no response of every other terminal to the particular query. Furthermore assume that because of reliability considerations it is undesirable to use a centralized algorithm at one terminal to conduct this query but rather a distributed algorithm which is simultaneously run at all the terminals must be used. Finally, assume that the network is in synchronism and that all terminals know of the response of all other terminals.

The most straightforward method of accomplishing this task is via a round-robin polling technique whereby all of the terminals respond to the query in some pre-determined order using a time-division multiplexing technique. If we have N terminals we would require N time slots, one dedicated to each terminal. The time slot must be of sufficient duration to carry the response of a terminal. We now make two assumptions, the result of which is to render the round-robin technique inefficient. We first assume that a terminal indicates a negative response to a query by transmitting nothing at all. This method of indicating a negative response is quite common in a network, especially when radio silence is important. The second assumption is that very few of the terminals will respond positively to the query. Our aim here is to investigate alternative distributive schemes which are more efficient than the round-robin scheme when these two assumptions hold.

The basic approach is to break the set of terminals into subsets and to query simultaneously all terminals in a subset. Then if no transmission is received from any terminal in the subset, all terminals in that subset are known to have responded negatively. If, however, one or more positive responses are received, further queries of the terminals in that subset in general are required. The querying is done by further subdividing the terminals in that subset into smaller subsets. All terminals in the network are able to know which subset is being queried at any time since they all receive all responses and can use these responses to drive a common algorithm which prescribes exactly which terminals are being queried. Thus no actual questions need be transmitted. Only the answers to the implied questions are transmitted over the communications channel.

Two different algorithms are explored in this paper. The first algorithm was originally suggested by Hayes (Hayes, J.F., 1978) and assumes that a terminal which desires to respond positively to a query transmits energy over the channel. If a group of terminals is simultaneously queried and energy appears on the channel in the slot allocated to the response, then all terminals know that at least one of the terminals in the subset queried answered affirmatively to the query. The details of this algorithm are described in the next section (Section 2) along with a sketch of the analysis of this algorithm.

The original Hayes algorithm asks some questions of groups of terminals, the answer to which could have been predicted before the questions were asked. These redundant questions can be skipped without any loss in performance. The subsequent section (Section 3) details a modification to the Hayes algorithm achieved by skipping redundant queries and an analysis of the improved algorithm.

The next section (Section 4) describes a new algorithm (Gudjohnsen, E. et al., 1980) for which fewer queries are required but for which more complicated answers are required. Now each terminal in the network is given a unique signature (or address) and if the terminal wishes to respond affirmatively it transmits its signature in the appropriate time slot. Now if a subset of the terminals is queried, if none or one of the terminals responds positively, the status of all terminals in the subset can be determined. (If one responds, the identity of that one can be determined by reading its signature—all others have a negative response.) Furthermore if two or more terminals simultaneously respond we assume that the signatures of all transmissions are garbled but that all receivers recognize that a garbled set of signatures was received so that they know there were two or more positive responses in the subset. In such a case, if the subset contains more than two terminals, a further subdivision is required. If the subset contains exactly two terminals no further subdivision is required since the garbled response must have been the result of both terminals transmitting their signatures. Various analyses are performed for this system.

First the average number of responses is calculated. Then the average number of bits in these responses is calculated using two different approaches.

In the sections to follow we will make the following common assumptions:

(1) The number of terminals N is a power of 2: i.e., $N = 2^k$. Thus where signatures are assigned, each signature is k bits long.

(2) For every terminal, the probability that the terminal wishes to respond positively is given by the parameter p , $0 \leq p \leq 1$. (Note that p is assumed the same for each terminal.)

(3) The random variables describing the responses of all N terminals to any query are statistically independent. Thus, the probability that exactly i of the N terminals wish to respond positively to a query is given by the formula

$$\binom{N}{i} p^i (1-p)^{N-i} \text{ for } i = 0, 1, 2, \dots, N.$$

To illustrate the steps followed in each of the algorithms we consider the following common example. Assume there are 16 terminals denoted $(0, 1, 2, \dots, 15)$. To a particular query, terminals 1, 10 and 11 wish to respond positively and all other terminals choose to respond negatively by preserving radio silence. For convenience, we show in Figure 1 all 16 terminals as the leaf nodes of a binary tree. These nodes are identified by the symbols, $0, 1, \dots, 15$ while the internal nodes are identified by the letters A, B, \dots, Q (with I and O omitted to avoid confusion with the integers 1 and 0). The asterisks next to leaf nodes 1, 10 and 11 indicate that they respond positively. All other leaf nodes respond negatively.

2. THE HAYES ALGORITHM (Hayes, J.F., 1978)

Hayes described two different versions of his algorithm which he termed non-adaptive and adaptive. We begin with a discussion of the non-adaptive version, since although the adaptive version is important from a practical standpoint, its understanding follows easily from the non-adaptive case.

As in the example depicted in Figure 1, the $N = 2^k$ terminals are identified with the 2^k leaves of a binary tree of depth k . A query is initially made of all the terminals by querying all of the leaf nodes that stem from the root node. (This is node A in Figure 1.) If all terminals respond negatively the algorithm is complete. If at least one of the terminals respond positively, then a query is made of terminals which stem from the node whose leaves are those in the upper half of the tree. (This is the node B in Figure 1.) If all terminals in this subset respond negatively the terminals corresponding to leaf nodes in the lower half of the tree are then queried. (This is node C in Figure 1.) Whenever a query of 2^{k-1} leaf nodes ($k > 1$) produces a positive response, the 2^{k-1} nodes are subdivided into two sets of 2^{k-2} nodes and each is queried separately. The process is iterated until, finally, individual leaf nodes are queried and the responses of all terminals are determined.

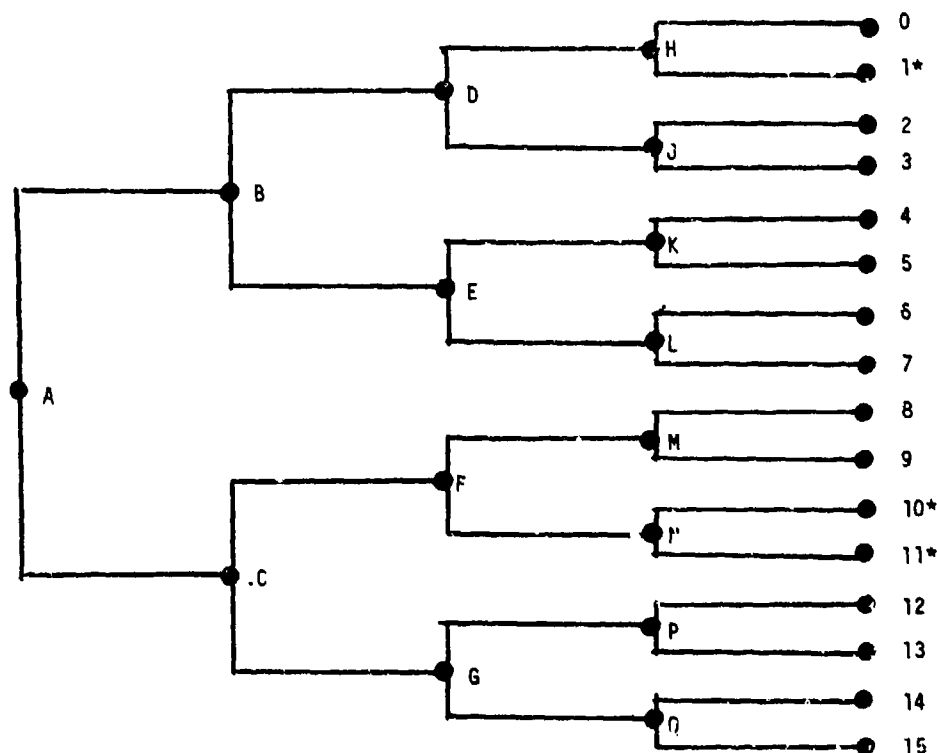


Figure 1. A common example for all algorithms—16 terminals with terminals 1, 10 and 11 responding positively.

Since the algorithm is known to all terminals, and since the responses to the queries are available to all terminals, no questions need to be asked. Rather the terminals respond to the next implicit question in the algorithm without any time (or bits) being wasted by actually asking the questions. The response to each implicit query only involves the terminals queried sending one bit of information.

To illustrate this algorithm, consider the examples of the 16 terminal network numbered (0,1,...,15) shown in Figure 1 where terminals 1, 10 and 11 wish to respond positively and all other terminals wish to respond negatively. For each implicit question in the algorithm, the following table contains the node in the tree from which the subtree grows, the leaf nodes (or terminals) which are being queried on each question, and the response which appears on the channel (yes or no).

Table 1

Queries and Responses for Example Given in Table 1 Using Hayes Algorithm			
Question No.	Node in Tree	Terminals Being Queried	Response
1	A	all	yes
2	B	0,1,2,3,4,5,6,7	yes
3	D	0,1,2,3	yes
4	H	0,1	yes
5	O	0	no
6	I	1	yes
7	J	2,3	no
8	E	4,5,6,7	no
9	C	8,9,10,11,12,13,14,15	yes
10	F	8,9,10,11	yes
11	M	8,9	no
12	N	10,11	yes
13	10	10	yes
14	11	11	yes
15	G	12,13,14,15	no

In this example, 15 queries were required to determine the responses of all 16 nodes. Round-robin polling would have required 16 queries so the savings here were not impressive. In order to determine what savings (if any) this algorithm achieves over round-robin polling, one requires either a mathematical analysis of the algorithm or a simulation. Fortunately, this algorithm admits to a neat mathematical analysis, the details of which are given in the Appendix. The end result is a recursive formula which gives the average number of queries required for a network of 2^k terminals, $E[Q_k(p)]$, in terms of the average number of queries for a terminal with 2^{k-1} terminals, $E[Q_{k-1}(p)]$. This formula is

$$E[Q_k(p)] = 2E[Q_{k-1}(p)] + 1 - 2(1-p)^{2^k} \quad k \geq 1$$

The initial condition to begin this recursive calculation is $E[Q_0(p)] = 1$ since it requires only one query to poll a network with a single terminal irrespective of the value of p . Numerical results for $E[Q_k(p)]$ for $k = 1$ to 10 and $p = .1$ to $.5$ in steps of $.1$ are given in Table II.

Table II

Unmodified Hayes

p	k	$E[Q_k(p)]$	$E[Q_k(p)]/2^k$
.1	1	1.380	.690
.1	2	2.488	.612
.1	3	5.035	.629
.1	4	10.699	.669
.1	5	22.329	.698
.1	6	45.655	.713
.1	7	92.310	.721
.1	8	185.621	.725
.1	9	372.247	.727
.1	10	745.483	.728
.2	1	1.720	.860
.2	2	3.621	.905
.2	3	7.906	.988

Table II (cont.)

p	k	$E[Q_k(p)]$	$E[Q_k(p)]/2^k$
.2	4	15.756	1.047
.2	5	34.510	1.078
.2	6	70.020	1.094
.2	7	141.040	1.102
.2	8	283.080	1.106
.2	9	567.161	1.108
.2	10	1135.322	1.109
.3	1	2.020	1.010
.3	2	4.560	1.140
.3	3	10.004	1.251
.3	4	21.002	1.313
.3	5	43.008	1.344
.3	6	87.008	1.359
.3	7	175.016	1.367
.3	8	351.031	1.371
.3	9	703.062	1.373
.3	10	1407.125	1.374
.4	1	2.280	1.140
.4	2	5.301	1.325
.4	3	11.568	1.446
.4	4	24.135	1.508
.4	5	49.271	1.540
.4	6	99.542	1.555
.4	7	200.084	1.563
.4	8	401.167	1.567
.4	9	803.334	1.569
.4	10	1607.669	1.570
.5	1	2.500	1.250
.5	2	5.875	1.469
.5	3	12.742	1.593
.5	4	26.484	1.655
.5	5	53.969	1.687
.5	6	108.937	1.702
.5	7	218.875	1.710
.5	8	438.750	1.714
.5	9	878.499	1.716
.5	10	1757.998	1.717

This completes the discussion of the original Hayes algorithm for the non-adaptive case. The essence of the adaptive case is the notion that for certain values of p it may be advantageous to treat 2^k terminals as two distinct sets of 2^{k-1} terminals (or four distinct sets of 2^{k-2} terminals, etc.) which are to be polled separately. In order to determine the optimum partitioning of the set of terminals we compute for each value of k and p the quantity $E[Q_k(p)]/2^k$. For a given value of p , we then denote by $k^*(p)$, the value of k for which $E[Q_k(p)]/2^k$ is a minimum. For a network with 2^k terminals, let $k_{OPT} = k^*$ if $k^* \leq k$. Otherwise let

$$k_{OPT} = \begin{cases} k & \text{if } k < k^* \\ k^* & \text{if } k \geq k^*. \end{cases}$$

Then one should partition the 2^k terminals into $2^{k-k_{OPT}}$ groups, each containing $2^{k_{OPT}}$ nodes and each group should be polled separately using the non-adaptive algorithm. The average number of queries required is then $2^{k-k_{OPT}} E[Q_{k_{OPT}}(p)]$. If $k^*(p) = 0$ the adaptive algorithm reduces to a round-robin algorithm. The values of $E[Q_k(p)]/2^k$ are also contained in Table II.

For example in Figure 1, if the 16 terminals were treated as 4 sets containing 4 terminals each, only 12 queries would be required. Similarly, 12 queries would be required if the 16 terminals were treated as 8 sets containing 2 terminals each.

3. THE MODIFIED HAYES SCHEME

The astute reader will have noticed that the Hayes algorithm, as described in the previous section, asks some questions to which the answers could have been predicted with certainty. Specifically, if 2^m leaf nodes are polled ($m \geq 1$), and a response is obtained, yet no response is obtained when the first half of the terminals are queried, it is certain that a positive response will be obtained when the second half are queried. Thus, these questions can be omitted from the algorithm with no loss of performance. (This latter statement assumes that all terminals reliably receive all responses. If errors can occur, these redundant queries and responses stabilize the algorithm.) The modified Hayes algorithm suggested here is thus to omit unnecessary questions. This modification can be used with either the non-adaptive or adaptive scheme.

For example of Figure 1 and Table I, using the non-adaptive scheme, queries 6 and 12 can be omitted since the answers to these questions are certainly "yes". Thus the number of queries, for this example, using the modified version of the non-adaptive scheme would be 13 instead of 15. It is left to the reader to count the queries for the modified version of the adaptive scheme.

It is desirable to know the average number of queries (or responses) required with the modified Hayes algorithm to poll 2^k terminals, each of which has a probability p of answering yes. Calling this quantity $E[Q_k(p)]$ the following recursive formula can be derived:

$$E[Q_k(p)] = 2E[Q_{k-1}(p)] + 1 - (1-p)^{2^{k-1}} - (1-p)^{2^k}, \quad k \geq 1.$$

Again the initial condition is $E[Q_0(p)] = 1$. Table III gives numerical results for the average number of queries as well as the information required in order to determine the best adaptive scheme.

Table III
Modified Hayes

p	k	$E[Q_k(p)]$	$E[Q_k(p)]/2^k$
.1	1	1.290	.645
.1	2	2.114	.528
.1	3	4.141	.518
.1	4	8.667	.542
.1	5	18.114	.566
.1	6	37.192	.581
.1	7	75.383	.589
.1	8	151.766	.593
.1	9	304.531	.595
.1	10	610.062	.596
.2	1	1.560	.780
.2	2	3.070	.768
.2	3	6.563	.820
.2	4	13.931	.871
.2	5	28.833	.901
.2	6	58.665	.917
.2	7	118.330	.924
.2	8	237.660	.928
.2	9	476.321	.930
.2	10	953.641	.931
.3	1	1.810	.905
.3	2	3.890	.972
.3	3	8.482	1.060
.3	4	17.903	1.119
.3	5	36.803	1.150
.3	6	74.606	1.166
.3	7	150.212	1.174
.3	8	301.423	1.177
.3	9	603.847	1.179
.3	10	1208.694	1.180
.4	1	2.040	1.020
.4	2	4.590	1.148
.4	3	10.034	1.254
.4	4	21.052	1.316
.4	5	43.103	1.347
.4	6	87.206	1.363
.4	7	175.413	1.370
.4	8	351.825	1.374
.4	9	704.651	1.376
.4	10	1410.302	1.377
.5	1	2.250	1.125
.5	2	5.188	1.297
.5	3	11.309	1.414
.5	4	23.613	1.476
.5	5	48.227	1.507
.5	6	97.453	1.523
.5	7	195.906	1.531
.5	8	392.812	1.534
.5	9	786.624	1.536
.5	10	1574.249	1.537

4. NEW ALGORITHM (Gudjohnsen, E. et al., 1980)

In this section we describe a new algorithm which, in general, requires fewer queries than the Hayes algorithm (even when modified) but requires more bits in each response. Several methods of evaluating the performance of this algorithm will be described. For each method, we will compare the results of this analysis with similar results for the Hayes algorithm. This algorithm is based upon a method originally put forth by Capetanakis (Capetanakis, J.I., 1979) for random access. We differ from Capetanakis in that he was concerned with the terminals sending a multi-bit message whereas we are concerned with the terminals only reporting their response to a single yes/no question. Furthermore, the focus of Capetanakis's work was on the situation with an infinite number of users whereas we are concerned with the case of a finite number of terminals.

In this algorithm each of the 2^k terminals is assigned a unique k bit signature. If the terminal wishes to respond positively it emits its signature. Again subsets of terminals are queried. If no signatures are imposed on the channel in response to a query of a subset of terminals, all terminals know that the response of all queried terminals in that subset is known. It is only when two or more impose their signature in response to a query that further queries are required. The exception to this latter statement is if only two terminals are queried. Then no matter what the response, the responses of these terminals are known by all.

Again an adaptive and non-adaptive version of this algorithm can be envisioned. We describe the non-adaptive version first. A query is initially asked of all 2^k terminals in the network. If none of the terminals respond or one of the terminals respond by transmitting its k bit signature the algorithm is complete. The algorithm is also complete if $k = 1$ irrespective of the response. If, however, for $k \geq 2$, two or more terminals respond by transmitting their signatures, the 2^k terminals are subdivided into 2 subsets containing 2^{k-1} terminals each and the process is repeated for each of the subsets until all of the responses are known. Questions which provide no new information are skipped just as in the modified Hayes algorithm.

This algorithm again can be thought of in terms of querying leaf nodes of a binary tree stemming from given internal nodes of the tree. The queries and responses for the example given in Figure 1 when this algorithm is employed are given in Table IV. The signature of the i th user is assumed to be the 4 bit binary representation of the decimal number (i.e., $0 \rightarrow 0000$, $1 \rightarrow 0001$, ..., $15 \rightarrow 1111$). Furthermore XXXX is used to denote the response when two or more signatures are transmitted, and ϕ is used to denote no response.

Table IV

Queries and Responses for Example Given in Table 1 Using New Algorithm

<u>Question Number</u>	<u>Node in Tree</u>	<u>Terminals Being Queried</u>	<u>Response</u>
1	A	all	XXXX
2	B	0,1,...,7	0001
3	C	8,9,...,15	XXXX
4	F	8,9,10,11	XXXX
5	M	8,9	ϕ
6	G	12,13,14,15	ϕ

Note that after two or more signatures were found as a response to query number 4 and no signatures were found as a response to query number 5, all terminals knew that terminals 10 and 11 responded positively (and 8 and 9 responded negatively).

Let $E[Q_k^n(p)]$ denote the average number of queries required by this algorithm to poll 2^k terminals, each of which had probability p of responding positively. The recursive formula which determines this quantity is

$$E[Q_k^n(p)] = 2E[Q_{k-1}^n(p)] + 1 - (1-p)^{2^k} - (1-p)^{2^{k-1}} - 3 \cdot 2^{k-1} p(1-p)^{(2^k-1)},$$

for $k \geq 2$. The initial condition here is $E[Q_1^n(p)] = 1$ since we need exactly one query to poll two terminals using this algorithm. Numerical values for $E[Q_k^n(p)]$ are given in Table V.

Table V

New Scheme—Count of Queries

<u>p</u>	<u>k</u>	<u>$E[Q_k^n(p)]$</u>	<u>$E[Q_k^n(p)]/2^k$</u>
.1	1	1	.5
.1	2	1.096	.274
.1	3	1.532	.192
.1	4	2.955	.185
.1	5	6.507	.203
.1	6	13.967	.218
.1	7	28.932	.226
.1	8	58.864	.230
.1	9	118.728	.232
.1	10	238.455	.235
.2	1	1	.5
.2	2	1.336	.334
.2	3	2.591	.324
.2	4	5.818	.364
.2	5	12.597	.394
.2	6	26.194	.409
.2	7	53.387	.417
.2	8	107.774	.421
.2	9	216.549	.423
.2	10	434.097	.424
.3	1	1	.5
.3	2	1.652	.413
.3	3	3.711	.464
.3	4	8.326	.520

Table V (cont.)

p	k	$E[Q_k(p)]$	$E[Q_k(p)]/2^k$
.3	5	17.649	.552
.3	6	36.298	.567
.3	7	73.597	.575
.3	8	148.194	.579
.3	9	297.388	.581
.3	10	595.775	.582
.4	1	1	.5
.4	2	1.992	.498
.4	3	4.703	.588
.4	4	10.385	.649
.4	5	21.769	.680
.4	6	44.539	.696
.4	7	90.078	.704
.4	8	181.156	.708
.4	9	363.312	.710
.4	10	727.623	.711
.5	1	1	.5
.5	2	2.313	.578
.5	3	5.512	.689
.5	4	12.019	.751
.5	5	25.038	.782
.5	6	51.077	.798
.5	7	103.153	.806
.5	8	207.306	.810
.5	9	415.613	.812
.5	10	832.225	.813

Also given in Table V is the value of $E[Q_k(p)]/2^k$, the quantity needed to determine the optimum partitioning for the adaptive version of the algorithm. (It is left up to the reader to fill in the details. They follow exactly as for the Hayes algorithm.)

Although the new algorithm requires fewer queries than does the Hayes algorithm (even when modified) the responses for the Hayes algorithm are only 1 bit long while here more bits are required in the responses. In order to compare bits in the responses we note that since all terminals know which subset is being queried, the terminals than know part of the signature of the potential responses. Only the portion of the signature which is not common to all potential responders need be transmitted. For example, referring to Table III, the response to question number 2 could have been 001 instead of 0001 since the signatures of all terminals being queried by this question began with a leading 0. Using this technique to count bits in the response, we define $E[B_k(p)]$ as the average number of bits in all responses by 2^k terminals each of which has probability p of responding positively. The recursive formula for $E[B_k(p)]$ can then be shown to be

$$E[B_k(p)] = 2E[B_{k-1}(p)] + k - (1-p)^{2^{k-1}}(k-1 + (2^{k-1}-1)p) + (5k-3)(2^{k-1}-1)p(1-p) + k((1-p)^{2^k} + 2^k p(1-p)^{2^k-1})$$

for $k \geq 2$ with initial condition $E[B_1(p)] = 1$. A further saving can be achieved by assuming that if no terminals respond, this lack of response can be recognized in 1 bit time. Defining $E[B'_k(p)]$ as the average number of bit-times required for such a system, one can derive the recursive formula.

$$E[B'_k(p)] = 2E[B'_{k-1}(p)] + k + (1-p)^{2^k} + k 2^k p (1-p)^{2^k-1} - (1-p)^{2^{k-1}}(k-1 + 3(1-p)^{2^{k-1}} + (3k+1)2^{k-1}p(1-p)^{2^{k-1}-1}),$$

for $k \geq 2$ again with initial condition $E[B'_1(p)] = 1$. Numerical values of these quantities are given in Table VI.

Let us compare the new algorithm with the modified Hayes algorithm by comparing $E[B'_k(p)]$ in Table VI with $E[Q'_k(p)]$ in Table III. If we only consider the non-adaptive version of both algorithms we find that for a fixed value of p (say $p = .1$), the new algorithm requires fewer bits in the response for small values of k while the modified Hayes algorithm requires fewer bits in the response for large values of k . If however we use the adaptive version of both algorithms we find that the new algorithm outperforms the modified Hayes algorithm.

Table VI

New Scheme-Bit Count

p	k	$E[B_k(p)]$	$E[B'_k(p)]$
.1	1	1	1
.1	2	2.096	1.440
.1	3	3.872	2.942
.1	4	8.414	7.399
.1	5	20.217	18.622
.1	6	46.194	43.041
.1	7	99.781	93.075

Table VI (cont.)

p	k	$E[B_k(p)]$	$E[B'_k(p)]$
.1	3	206.762	194.150
.1	9	422.524	397.301
.1	10	855.048	804.607
.2	1	1	1
.2	2	2.336	1.926
.2	3	5.511	5.027
.2	4	13.927	13.213
.2	5	32.700	31.293
.2	6	71.396	68.581
.2	7	149.791	144.162
.2	8	307.583	296.324
.2	9	624.166	601.648
.2	10	1258.331	1213.296
.3	1	1	1
.3	2	2.652	2.412
.3	3	7.117	6.834
.3	4	17.948	17.431
.3	5	40.881	39.849
.3	6	87.762	85.698
.3	7	182.524	178.396
.3	8	373.048	364.793
.3	9	755.096	738.585
.3	10	1520.193	1487.171
.4	1	1	1
.4	2	2.992	2.862
.4	3	8.422	8.253
.4	4	20.780	20.447
.4	5	46.559	45.893
.4	6	99.118	97.787
.4	7	205.237	202.573
.4	8	418.473	413.146
.4	9	845.947	835.293
.4	10	1701.894	1680.586
.5	1	1	1
.5	2	3.313	3.250
.5	3	9.398	9.305
.5	4	22.784	22.597
.5	5	50.558	50.194
.5	6	107.136	106.388
.5	7	221.272	219.776
.5	8	450.544	447.552
.5	9	910.087	904.104
.5	10	1830.175	1818.207

APPENDIX

We consider the unmodified Hayes Scheme described in Section 2. We assume we have a tree consisting of 2^k leaf nodes, the upper subtree with 2^{k-1} leaf nodes and the lower subtree with 2^{k-1} nodes. Consider the following four mutually exclusive and exhaustive events:

E_1 : No positive responses from 2^k terminals.

E_2 : No positive responses from upper 2^{k-1} terminals; one or more positive responses from lower 2^{k-1} terminals.

E_3 : No positive responses from lower 2^{k-1} terminals; one or more positive responses from upper 2^{k-1} terminals.

E_4 : One or more positive responses from both lower and upper set of 2^{k-1} terminals.

The probability of these four events are:

$$P(E_1) = (1-p)^{2^k},$$

$$P(E_2) = P(E_3) = (1-p)^{2^{k-1}} (1 - (1-p)^{2^{k-1}}),$$

$$P(E_4) = (1 - (1-p)^{2^{k-1}})^2.$$

Let $E[Q_k(p)]$ be the average number of queries required to poll the 2^k terminals. Then

$$E[Q_k(p)] = \sum_{i=1}^4 E[Q_k(p) | E_i] P(E_i).$$

But

$$E[Q_k(p)|E_1] = 1,$$

$$E[Q_k(p)|E_2] = E[Q_k(p)|E_3] = 2 + E[Q_{k-1}(p)|E_5],$$

$$E[Q_k(p)|E_4] = 1 + 2E[Q_{k-1}(p)|E_5],$$

where E_5 is the event that there are one or more positive responses from a set of 2^{k-1} terminals.

But it is easy to verify that

$$E[Q_{k-1}(p)|E_5] = \frac{E[Q_{k-1}(p)] - (1-p)^{2^{k-1}}}{1 - (1-p)^{2^{k-1}}}.$$

Substituting we then find that

$$E[Q_k(p)] = 1 + 2E[Q_{k-1}(p)] - 2(1-p)^{2^k}$$

which is the desired result.

Similar derivations yield the other recursive formulas given in this paper.

ACKNOWLEDGEMENT

This work was supported by the National Science Foundation under Grant EEC-7921140. Much of this work was done in collaboration with Professor Don Towsley and the author gratefully acknowledges his permission to publish this work here.

REFERENCES

- Capetanakis, J. I., September 1979, "Tree Algorithms for Packet Broadcast Channels," IEEE Trans. on Information Theory, Vol. IT-25, pp. 505-515.
- Gudjohnsen, E., D. Towsley and J. K. Wolf, June 1980, "On Adaptive Polling Techniques for Computer Communication Networks," ICC Conference Record, Vol. 1, pp. 13.3.1-13.3.5.
- Hayes, J. F., August 1978, "An Adaptive Technique for Load Distribution," IEEE Trans. on Communications, Vol. COM-26, pp. 1178-1186.

DISCUSSIONS
SESSION IV

REFERENCE NO. OF PAPER: IV-15

DISCUSSOR'S NAME: Harvey Nelson, Naval Weapons Center, USA

AUTHOR'S NAME: A. O. Ward

COMMENT: I'm very interested in your concept of an engineer work station. That is an integrated net of automated tools for developing requirements and proceeding on to PSL/PSA and code. When do you expect to have the work station operational? What will be the steps of implementation? What marketing or availability plans do you foresee?

AUTHOR'S REPLY: We hope to have the work station operational in the first half of 1982 but I would prefer not to detail the implementation steps here. The tool is being developed for in-house use and we do not have any short-term plans to make it commercially available.

REFERENCE NO. OF PAPER: IV-15

DISCUSSOR'S NAME: Dr. N. J. B. Young, Ultra Electronic Controls

AUTHOR'S NAME: A. O. Ward

COMMENT: Have you considered producing code automatically in a second language, e.g. PASCAL, as well as in CORAL? A second code implementation would enable you to perform a dual-coding type test of the first implementation (but would not help in checking the specification, of course).

AUTHOR'S REPLY: We have considered languages other than CORAL but not for the reason you cite. Taking a long view, we will certainly wish to have a similar capability for use with the ADA language and a program of work is being considered to achieve this.

REFERENCE NO. OF PAPER: IV-15

DISCUSSOR'S NAME: Dr. von Issendorff

AUTHOR'S NAME: A. O. Ward

COMMENT: I am quite impressed by your method which seems to be very usable. But, in case you would like to select your method or another one--and there are many more--I would not have the means to do so. So, could you please compare your method to others.

AUTHOR'S REPLY: To answer this question properly is clearly outside the scope of this meeting. So, I would like to respond in two ways. First, when we were formulating our ideas on requirements analysis just over 2 years ago, there seemed to be few alternatives. TRW's RSL/REVS system, although powerful, was not commercially available and the host machine and language were not compatible with our environment. SADT was not widely accessible in the United Kingdom and we understood that efforts to model SADT descriptions in PSL had not proved successful at that time.

As far as tools were concerned, there were two alternatives, Michigan's PSL/PSA and the U.K. system SDS. The latter required significant front-end effort to be made practicable and again was only available on a host-machine to which we did not have access. PSL/PSA, on the other hand had a rich language and was supported on our mainframes.

The second point I would make is that there are two studies which Dr. von Issendorff may find useful. The first was sponsored by RSRE and is in the public domain, being an international survey of requirements analysis methods and tools. The second is currently being undertaken by the Department of Industry and is entitled "DoI Ada Methodology Study." The latter should report before the end of 1981.

REFERENCE NO. OF PAPER: IV-17

DISCUSSOR'S NAME: K. Brammer, ESG

AUTHOR'S NAME: Enslow (Livesey, presenter)

COMMENT: Would you explain how priority interrupts/requests are handled by the fully distributed processing system (where the participating units seem to have equal rights); for instance, if a five control component within an avionics system needs instant action. Can you elaborate on the notion of the "price," a user of the FDPS has to offer while bidding for being served. Is it meant literally (in dollars) or is it an abstract concept?

AUTHOR'S REPLY: (1) Components can have equal rights in the sense of cooperative autonomy, but still have differing priorities. If a user in the system needs (and deserves) instant service, then it is effectively bidding a very high "price" and should win most contests for resources.

An extremely high priority user might even have dedicated resources.

(2) The price in bidding is whatever is meaningful in the system: dollars, budget, or priorities, etc.

(This is the opinion of the presenter, not necessarily that of the author.)

REFERENCE NO. OF PAPER: IV-17

DISCUSSOR'S NAME: T. Smeistrd, NDRE, Norway

AUTHOR'S NAME: Enslow (Livesey, presenter)

COMMENT: When trying to increase the performance of decentralized decision making, one is often faced with the second-guessing phenomenon (a decision maker anticipates the actions of other decision makers to make his own actions more effective). Is this phenomenon present in your problem formulations - and can it be used to improve the performance?

AUTHOR'S REPLY: One classic example of this is the bidding problem. One asks for a resource. One is told it is available, but then when one reserves the resource, it has already been taken by someone else. This is due to time delays in inquiries and reservations. At Georgia Tech, we are actively investigating this problem.

(This is the opinion of the presenter, not necessarily that of the author.)

REFERENCE NO. OF PAPER: IV-18

DISCUSSOR'S NAME: Enslow (Livesey)

AUTHOR'S NAME: L. Svobodova, INRIA

COMMENT: Is it not true that in this system message passing would have to be atomic (if a crash occurred after the textual part of a message arrived, but before the message identifier did (or vice-versa) then an inconsistent state might result)? Do you know any system in which this is taken care of?

AUTHOR'S REPLY: (1) It is not necessary that the communication subsystem delivers messages atomically, however, the receiver must be able to check the integrity of a request. If the textual part of a message arrived before the identifier of the atomic action to be created by the request, the request would not be processed, since the first thing that must be done is to create a commit record, for which it is necessary to have the identifier. If the textual part got lost, the atomic action opened when the identifier was received would be aborted, since a timeout is associated with each commit record. (2) Communication protocols that provide virtual connections deliver messages atomically, however, it does not mean yet that a message is delivered atomically to the destination process. I do not know any distributed system that implements atomic process to process communication.

REFERENCE NO. OF PAPER: IV-18

DISCUSSOR'S NAME: Van Keuk, AVP Member

AUTHOR'S NAME: L. Svobodova

COMMENT: Crashes as you said can occur as a consequence of incorrect stochastic data as we are faced with in signal processing applications. Do you see a conflict of your technique and backtracing facilities for test and debugging.

AUTHOR'S REPLY: It is true that automatic rollback to an earlier state conflicts with testing and debugging, where it is important to keep a trace also of the erroneous states. However, the mechanisms that I described are intended to facilitate orderly recovery rather than impose it at all times. That is, it would be possible to inhibit them while in a debugging stage. Also, we have been designing a system where the object versions, even the invalidated ones, are preserved for an unlimited period of time. The invalidated versions are not accessible to ordinary user programs, but they could be made available to a debugger. (See reference SV08 80)

REFERENCE NO. OF PAPER: IV-18

DISCUSSOR'S NAME: K. Shin, Rensselaer Polytechnic Institute, USA

AUTHOR'S NAME: L. Svobodova

COMMENT: Did you carry out any overhead analysis? Otherwise, how can you justify your proposed method?

AUTHOR'S REPLY: No, I did not do any overhead analysis of the method that I have described in my paper. Clearly, it is important to find out if this method is practical, however, I do not agree that what is needed is overhead analysis. Overhead with respect to what? I believe that the ease of developing reliable software offered by the method (a programmer can write application software without any concern about restoring consistent state in case of a failure) is more important than the additional processor time and memory need to implement it. And, I believe that only experimental work can demonstrate if the proposed method is practical.

REFERENCE NO. OF PAPER: IV-19

DISCUSSOR'S NAME: Horst Kister, Germany

AUTHOR'S NAME: J. K. Wolf

COMMENT: Why using polling method at all? Why not issuing a broadcast and let then all terminals with a "yes" respond in a priority order? (Any modern system should be able to do more than polling.)

AUTHOR'S REPLY: Since any terminal initially only knows its own state, the suggested technique of responding in priority order is equivalent to "roll-call" or "hub" polling which the paper shows is not as efficient as probing.

Also, I believe the author has in mind a system where the polling is used as a method of terminals gaining channel access to transmit information. This is only one of many uses to which polling can be put. Status collection is a different use.

REFERENCE NO. OF PAPER: IV-19

DISCUSSOR'S NAME: K. G. Shin, Rensselaer Polytechnic Institute, USA

AUTHOR'S NAME: Prof. Wolf

COMMENT: How would you handle an error in answering the query?

AUTHOR'S REPLY: Some of the algorithms are more sensitive to errors than others. As a rule of thumb, the more efficient the algorithms, the less redundancy exists in the algorithm and thus the more sensitive the algorithm is to errors.

REFERENCE NO. OF PAPER: IV-19

DISCUSSOR'S NAME: J. H. Saltzer, MIT, USA

AUTHOR'S NAME: Prof. Wolf

COMMENT: How about applying this polling technique to a speed-limit circuit? In that case, there may be a longer time required to poll a larger number of points because of fan-out. It would seem that this effect would lead to a different optimum polling pattern.

AUTHOR'S REPLY: This is an excellent suggestion. We have considered a problem which is in some sense the dual of the problem you suggest whereby there is an upper limit to the number of times a station can be "probed" in any given polling cycle. It certainly makes a great deal of sense to consider the problem you suggest.

REFERENCE NO. OF PAPER: IV-19

DISCUSSOR'S NAME: Dr. Van Keuk, AVP Member

AUTHOR'S NAME: Prof. Wolf

COMMENT: For your analysis you need, as you said, assumptions on the statistical independence of the events. I feel in addition you need the assumption of constant probability. If this is not given, how do you modify your algorithm.

AUTHOR'S REPLY: We are presently working on just this problem. We are considering the simplest case where we have two classes of stations, one class having probability P_1 of being active and the other class having probability P_2 of being active. At this time, I cannot give you any concrete results except to say that one must carefully match the algorithm to the assumptions on the statistics of the stations in order to achieve an efficient scheme.

STAGE-STATE RELIABILITY ANALYSIS TECHNIQUE

Alan D. Stern
Boeing Military Airplane Company
Digital Flight Controls Research
Seattle, Washington

SUMMARY

Conventional reliability analysis techniques such as fault-tree and Boolean algebra methods are difficult to apply to redundant systems with complex interactions and redundancy management philosophies. Some advanced flight control systems, for example, employ multiple redundant channels which, with proper redundancy management and failure detection, can degrade to simplex operation. The reliability analysis must properly account for the defined success criteria, redundancy level, redundancy management technique, system dependencies, and failure detection coverage. The Stage-State reliability analysis technique properly accounts for these factors. It is also computationally simple such that triplex redundant systems have been analyzed using an early 1970's desktop computer.

This method is well suited for analysis by the system architect. The process begins with a system block diagram showing all element connections. A success logic diagram is then written reflecting all possible success states. The probability of success equation is written directly from the logic diagram and evaluated by substituting the probability expression for each system element. Multiple success criteria can be applied to one problem formulation simply by deleting those states which do not satisfy the success criterion.

1.0 INTRODUCTION

Advanced digital flight control systems (DFCS) for new aircraft are assuming additional roles relative to today's operational vehicles. Such roles include stability augmentation systems (SAS) and maneuver load alleviation (MLA) systems plus other requirements which may require the DFCS to have flight safety reliability over significant portions of the flight envelope. Provision of such reliability while simultaneously striving to minimize hardware redundancy levels, have led to the development of sophisticated DFCS architectures. Some promising system architectures have included in their redundancy management philosophies, the ability to isolate failures to a particular line replaceable unit (LRU) and to select that LRU successfully to the simplex level. The ability to redundancy manage LRU's in this fashion requires that the architecture provide the transfer of data (from redundant LRU's) between channels (see Figure 1), and that the selection of one healthy LRU from two choices be achievable. The probability of selecting one healthy LRU when one of two redundant LRU's has failed is called "failure coverage" or just "coverage".

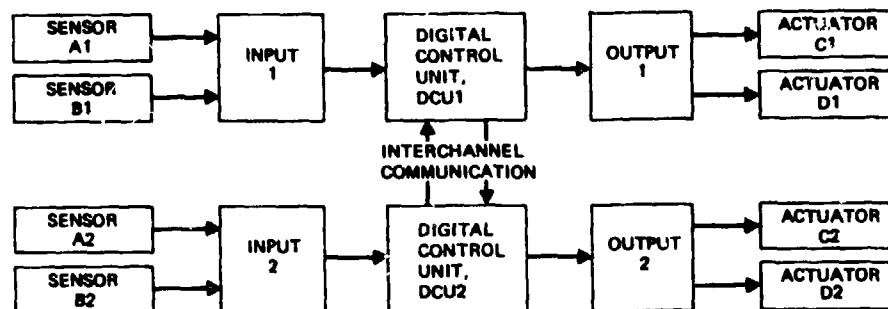


Figure 1. Duplex DFCS With Interchannel Communication

The concept of various degrees of "dependency" also arises with such architectures. An LRU has a dependency when it must rely upon one or more other LRU's to operate successfully before that LRU can accomplish its function in the system. For example, Figure 1 shows that the transfer of sensor data between channels depends upon successful operation of its Input and DCU.

Conventional reliability analysis techniques such as fault-tree and Boolean algebra methods, become extremely difficult to use for complex architectures possessing redundancy with dependencies and coverage. The mathematics becomes massive with high probability for error. The Stage-State reliability analysis technique, on the other hand, is quite simple while possessing the following features:

- accounts for redundancy level for each specific LRU,
- defines the collection of probability states which represent a desired success or failure criteria,
- a probability of success $P(S)$ equation can be written directly from a success logic diagram which includes the effects of dependencies,
- the $P(S)$ equation is more compact and requires minimal memory for a digital evaluation relative to competing methods,
- multiple success criteria are easily evaluated by simply deleting those states (terms) which do not satisfy the new success criteria, and
- the effect of failure coverage is easily incorporated in the $P(S)$ equation.

2.0 THE METHOD

2.1 Concept Definition

The Stage-State method, described below, was developed by Mr. Jimmy Rice of The Boeing Military Airplane Company in 1978 to support the systematic analysis of a variety of DFCS architectures with a large number of system elements (Reference 1). The need he fulfilled was to provide a reliability analysis tool that the system designer could easily use to conduct system architecture trade studies.

The method is based upon straight-forward use of set theory and axioms of probability. It considers a space S which contains all possible outcomes of the system and breaks them up into mutually exclusive events, or states. The sum of the probabilities of all such states must therefore be unity.

Consider the following example of a system consisting of a duplex stage. A "stage" is defined as a set of like redundant elements (LRU's) as

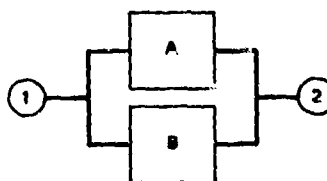


Figure 2. Duplex Stage

shown in Figure 2. This figure illustrates a success logic diagram; i.e., success consists of the chain between points 1 and 2 not being totally broken due to failures. This stage consists of four independent states. A "state" defines a particular combination of failed and/or healthy LRU's of a given stage. The possible duplex states are

$$\begin{array}{llll}
 \text{Both Healthy} & = R_A R_B = ST1 = \text{State 1} = P \left(\begin{array}{c} AB \\ DUP \end{array} \right) \\
 \text{"A" Healthy, "B" Failed} & = R_A Q_B = ST2 = \text{State 2} = P \left(\begin{array}{c} AB \\ DUP \end{array} \right) \\
 \text{"A" Failed, "B" Healthy} & = Q_A R_B = ST3 = \text{State 3} = P \left(\begin{array}{c} AB \\ DUP \end{array} \right) \\
 \text{Both Failed} & = Q_A Q_B = ST4 = \text{State 4} = P \left(\begin{array}{c} AB \\ DUP \end{array} \right)
 \end{array} \quad (1)$$

where R_i and Q_i are the probabilities that the i th LRU is good or bad, respectively. The probability that the stage is good or bad is

$$P(S) = 1.0 = R_A R_B + R_A Q_B + Q_A R_B + Q_A Q_B \quad (2)$$

A success criterion can be applied to these states. If that criterion is that either A or B good represents success, then

$$P(\text{success}) = R_A R_B + R_A Q_B + Q_A R_B \quad (3)$$

If success says that both must be good, then only state 1 applies; i.e., $P(\text{success}) = R_A R_B$.

The Stage-State technique employs conditional probability to adjust success criteria. Let S be defined as the success function. The probability of S occurring for a duplex stage is

$$P(S) = \sum_{i=1}^4 P(S/ST_i) P(ST_i) \quad (4)$$

where $P(S/ST_i)$ is the conditional probability of success given the stage is in state i (ST_i). For the duplex stage, where either element healthy constitutes success, equations (1) indicate

$$\begin{array}{l}
 P(S/ST1) = P(S/ST1) = P(S/ST2) = P(S/ST3) = 1 \\
 P(S/ST4) = 0
 \end{array} \quad (5)$$

Therefore, $P(S)$ for duplex stage AB is the probability that that stage is in states 1, 2, or 3 duplex.

$$P(S) = \overset{\text{STG AB}}{P(1,2,3)} = \underbrace{1}_{\text{DUP}} \underbrace{P(ST1)}_{R_A R_B} + \underbrace{1}_{\text{DUP}} \underbrace{P(ST2)}_{R_A Q_B} + \underbrace{1}_{\text{DUP}} \underbrace{P(ST3)}_{Q_A R_B} \quad (6)$$

If elements A and B are identical, then defining the reliability of LRU A as R_A and the probability of failure of A as Q_A we get

$$P(S) = R_A^2 + 2 R_A Q_A \quad (7)$$

$$\text{or} \quad = 2 R_A - R_A^2 \quad (8)$$

$$\text{where } Q_A = 1 - R_A \quad (9)$$

2.2 Effect of Dependencies

Consider the duplex system shown in Figure 3 which has three duplex stages (A, B, and C) with C being a dependency for stages A and B. Success is defined as getting information to the success node, S.

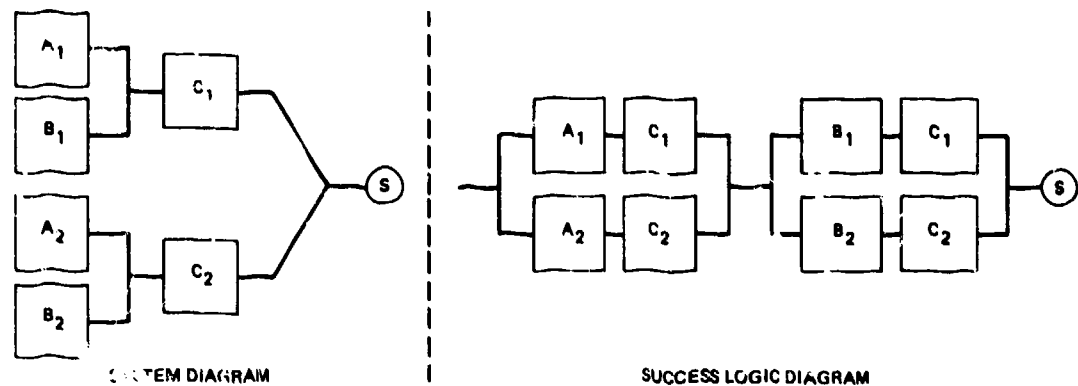


Figure 3. Duplex System With Dependency (Stage C)

Two success states will be defined for each stage - both LRU's good (ST1); and either LRU good (ST2). The probability of success can be written as

$$P(S) = \overset{\text{STGC}}{P(S/C)} \overset{\text{STGC}}{P(1)}_{\text{DUP}} + \overset{\text{STGC}}{P(S/C)} \overset{\text{STGC}}{P(2)}_{\text{DUP}} \quad (10)$$

This equation examines success based upon the most dependent stage first. It reads: $P(S)$ equals the probability success given stage C is in state 1 (duplex) times the probability that stage C is in state 1, plus the probability of success given stage C is in state 2 (duplex) times the probability that stage C is in state 2.

To examine the first term we can redraw Figure 3 assuming stage C is in state 1; i.e., both good.

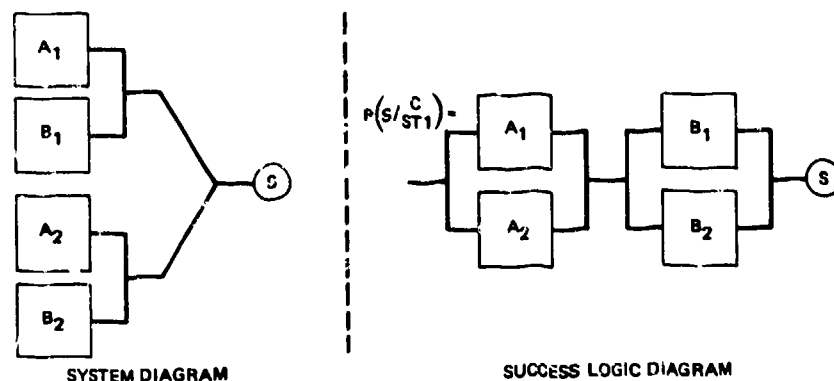


Figure 4. Duplex System With Stage C in State 1

From Figure 4 it is readily observed that the conditional success can now be defined as follows

$P(S/ST1)$ = Probability of A_1 or A_2 ; and B_1 or B_2 good

$$P(S/C) = \begin{matrix} \text{STG A} & \text{STG B} \\ P(1,2) & P(1,2) \\ \text{ST1} & \text{DUP} \end{matrix} \quad (11)$$

From equation (8) the following is written

$$= (2R_A - R_A^2)(2R_B - R_B^2) \quad (12)$$

Now let's evaluate the conditional probability where stage C is in state 2; i.e., only C_1 or C_2 good. Figure 5 illustrates this state.

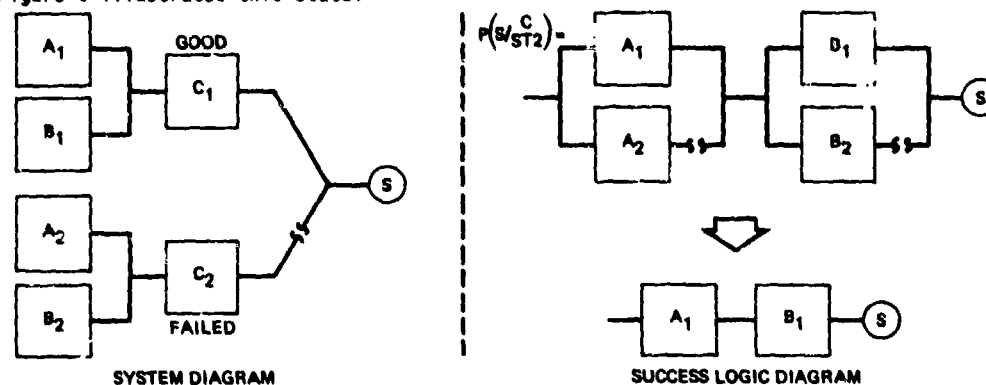


Figure 5. Duplex System With Stage C in State 2

From Figure 5 the probability for success given stage C is in state 2 is

$$P(S/ST2) = P(\text{STG A})_{\text{SIMP}} P(\text{STG B})_{\text{SIMP}} = R_A \times R_B \quad (13)$$

That is, if C_1 is healthy, success depends upon the probability that the elements A and B are healthy in their simplex state. Substituting (12) and (13) into (10) gives the final result.

$$P(S) = (2R_A - R_A^2)(2R_B - R_B^2) \underbrace{P(\text{STG C})_{\text{DUP}}}_{R_C^2} + R_A R_B \underbrace{P(\text{STG C})_{\text{DUP}}}_{2R_C Q_C = 2R_C - 2R_C^2} \quad (14)$$

$$P(S) = (2R_A - R_A^2)(2R_B - R_B^2) R_C^2 + R_A R_B (2R_C - 2R_C^2) \quad (15)$$

Equation (15) was derived with relative ease. A comparison with a Boolean algebra solution to the same problem is illustrated in Reference 1 which shows three pages of detailed algebra were required to obtain a result identical to the Stage-State method.

2.3 A More Complex Example

A duplex DFCS will now be evaluated. This system, shown in Figure 6, has duplex stages for all LRU's; i.e., sensors (A and B), input sections (IN), digital control units (DCU), output sections (OUT), and control surface servos (AIL, RUD, and ELE). The DCU's have an interchannel communication capability. This DFCS has three dependencies. The most dependent element is

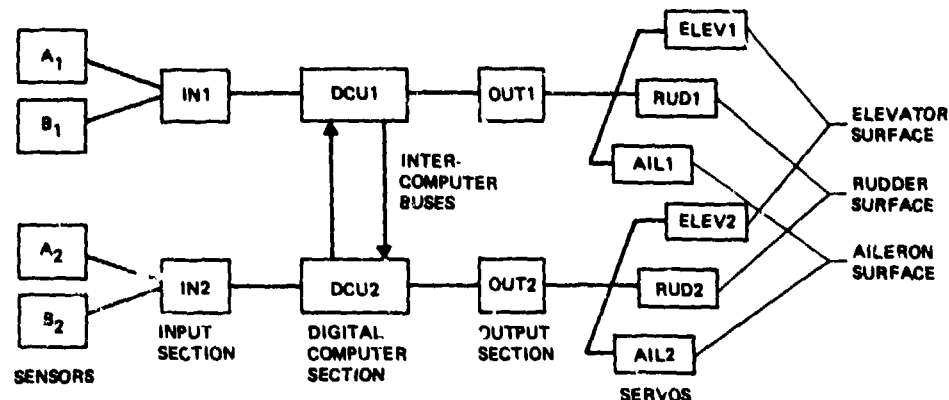


Figure 6. Dual DFCS

the DCU since its loss constitutes loss of a full channel. The input and output dependency impacts the use of one full sensor or servo set. This is illustrated by the associated success logic diagram shown in Figure 7. This

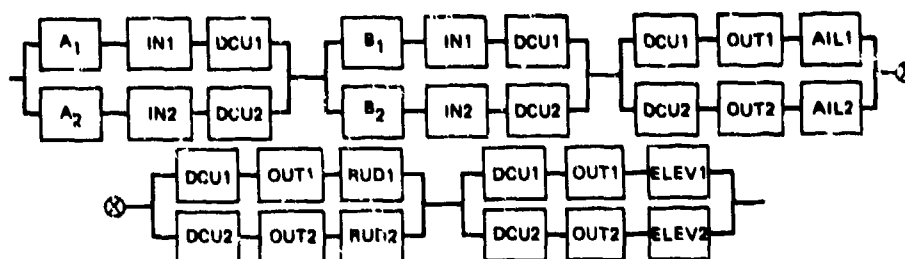


Figure 7. Success Logic Diagram for the Dual DFCS

logic diagram can be used to define success states subject to various levels of conditional probabilities as defined below.

The probability of system success $P(S)$ can be written using repetitive application of the definition of conditional probability and the sub-division of the sample space into disjoint states. For the DCU stage, $P(S)$ is

$$P(S) = P\left(\frac{DCU}{S/1}\right)P\left(\frac{DCU}{1}\right) + P\left(\frac{DCU}{S/2}\right)P\left(\frac{DCU}{2}\right) \quad (16)$$

For the DCU in state 1 duplex, the system reduces to that shown in Figure 8, and $P\left(\frac{DCU}{S/1}\right)$ is defined from this diagram.

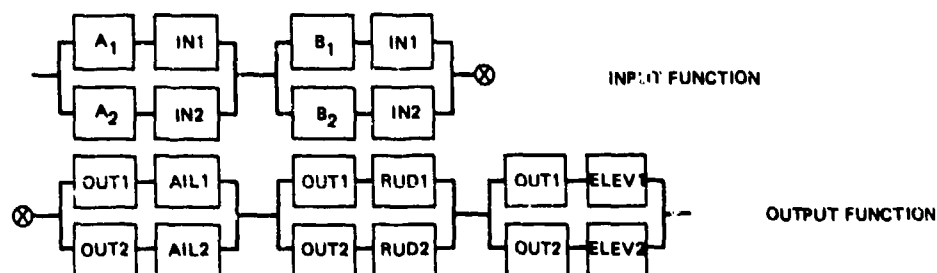


Figure 8. DFCS With DCU in State 1 (Both Good)

Observe that the first subdivision of the sample space was to divide it into all of the possible states of the most dependent stage. Given that the DCU stage is in state 1 duplex, the system is reduced to one composed of independent input and output functions but which possess internal dependencies. The input function has the input stage as a dependency and the output function has the output stage as a dependency. Then

$$\begin{aligned} P\left(\frac{DCU}{S/1}\right) &= P[(\text{INPUT FUNCTION GOOD}) \cdot (\text{OUTPUT FUNCTION GOOD})] \\ &= P(\text{INPUT FUNCTION GOOD}) \cdot P(\text{OUTPUT FUNCTION GOOD}) \\ &= P(\text{INF})P(\text{OUTF}) \end{aligned} \quad (17)$$

The input function is now subdivided into a reduced set of disjoint sample spaces.

$$P(\text{INF}) = P\left(\frac{\text{IN}}{\text{INF}/1}\right)P\left(\frac{\text{IN}}{1}\right) + P\left(\frac{\text{IN}}{\text{INF}/2}\right)P\left(\frac{\text{IN}}{2}\right) \quad (18)$$

and

$$P\left(\frac{\text{IN}}{\text{INF}/1}\right) = \begin{array}{c} \text{A}_1 \\ \text{A}_2 \end{array} \begin{array}{c} \text{B}_1 \\ \text{B}_2 \end{array} = P\left(\frac{\text{A}}{1,2}\right)P\left(\frac{\text{B}}{1,2}\right) \quad (19)$$

$$\begin{aligned}
 P(\text{INF}/2)_{\text{DUP}} &= \begin{array}{c} \text{A}_1 \\ \text{A}_2 \end{array} \begin{array}{c} \text{B}_1 \\ \text{B}_2 \end{array} \begin{array}{c} \text{IN}_1 \\ \text{IN}_2 \end{array} = \begin{array}{c} \text{A}_1 \\ \text{A}_2 \end{array} \begin{array}{c} \text{B}_1 \\ \text{B}_2 \end{array} \begin{array}{c} \text{IN}_1 \\ \text{IN}_2 \end{array} \\
 &= P(\frac{A}{1})_{\text{SIMP}} P(\frac{B}{1})_{\text{SIMP}} P(\frac{\text{IN}}{1})_{\text{SIMP}} \quad (20)
 \end{aligned}$$

therefore,

$$P(\text{INF}) = P(\frac{A}{1,2})_{\text{DUP}} P(\frac{B}{1,2})_{\text{DUP}} P(\frac{\text{IN}}{1})_{\text{DUP}} + P(\frac{A}{1})_{\text{SIMP}} P(\frac{B}{1})_{\text{SIMP}} P(\frac{\text{IN}}{2})_{\text{DUP}} \quad (21)$$

In a similar fashion the output function is subdivided into its reduced sample space.

$$P(\text{Output Function is Good}) = P(\text{OUTF})$$

$$P(\text{OUTF}) = P(\frac{\text{OUT}}{\text{OUTF}/1})_{\text{DUP}} P(\frac{\text{OUT}}{1})_{\text{DUP}} + P(\frac{\text{OUT}}{\text{OUTF}/2})_{\text{DUP}} P(\frac{\text{OUT}}{2})_{\text{DUP}} \quad (22)$$

where

$$\begin{aligned}
 P(\frac{\text{OUT}}{\text{OUTF}/1})_{\text{DUP}} &= \begin{array}{c} \text{AIL1} \\ \text{AIL2} \end{array} \begin{array}{c} \text{RUD1} \\ \text{RUD2} \end{array} \begin{array}{c} \text{ELEV1} \\ \text{ELEV2} \end{array} \\
 &= P(\frac{\text{AIL}}{1,2})_{\text{DUP}} P(\frac{\text{RUD}}{1,2})_{\text{DUP}} P(\frac{\text{ELE}}{1,2})_{\text{DUP}} \quad (23)
 \end{aligned}$$

and

$$\begin{aligned}
 P(\frac{\text{OUT}}{\text{OUTF}/2})_{\text{DUP}} &= \begin{array}{c} \text{AIL1} \\ \text{AIL2} \end{array} \begin{array}{c} \text{RUD1} \\ \text{RUD2} \end{array} \begin{array}{c} \text{ELEV1} \\ \text{ELEV2} \end{array} \begin{array}{c} \text{OUT1} \\ \text{OUT2} \end{array} \\
 &= P(\frac{\text{AIL}}{1})_{\text{SIMP}} P(\frac{\text{RUD}}{1})_{\text{SIMP}} P(\frac{\text{ELE}}{1})_{\text{SIMP}} P(\frac{\text{OUT}}{1})_{\text{SIMP}} \quad (24)
 \end{aligned}$$

substituting (23) and (24) into (22) and then (22) and (21) into (17) gives

$$\begin{aligned}
 P(\frac{\text{DCU}}{\text{S}/1})_{\text{DUP}} &= \left[P(\frac{A}{1,2})_{\text{DUP}} P(\frac{B}{1,2})_{\text{DUP}} P(\frac{\text{IN}}{1})_{\text{DUP}} + P(\frac{A}{1})_{\text{SIMP}} P(\frac{B}{1})_{\text{SIMP}} P(\frac{\text{IN}}{2})_{\text{DUP}} \right] \times \\
 &\quad \left[P(\frac{\text{AIL}}{1,2})_{\text{DUP}} P(\frac{\text{RUD}}{1,2})_{\text{DUP}} P(\frac{\text{ELEV}}{1,2})_{\text{DUP}} P(\frac{\text{OUT}}{1})_{\text{DUP}} + P(\frac{\text{AIL}}{1})_{\text{SIMP}} P(\frac{\text{RUD}}{1})_{\text{SIMP}} P(\frac{\text{ELEV}}{1})_{\text{SIMP}} P(\frac{\text{OUT}}{2})_{\text{DUP}} \right] \quad (25)
 \end{aligned}$$

Looking at the conditional probability $P(\frac{\text{DCU}}{\text{S}/2})_{\text{DUP}}$ from equation (16), it can be seen that the success path is now one simplex channel so that

$$P(\frac{\text{DCU}}{\text{S}/2})_{\text{DUP}} = P(\frac{A}{1})_{\text{SIMP}} P(\frac{B}{1})_{\text{SIMP}} P(\frac{\text{IN}}{1})_{\text{SIMP}} P(\frac{\text{OUT}}{1})_{\text{SIMP}} P(\frac{\text{AIL}}{1})_{\text{SIMP}} P(\frac{\text{RUD}}{1})_{\text{SIMP}} P(\frac{\text{ELEV}}{1})_{\text{SIMP}} \quad (26)$$

Substitution of equations (25) and (26) into (16) provides the final result.

$$\begin{aligned}
 P(\text{S}) &= \left\{ P(\frac{A}{1,2})_{\text{DUP}} P(\frac{B}{1,2})_{\text{DUP}} P(\frac{\text{IN}}{1})_{\text{DUP}} + P(\frac{A}{1})_{\text{SIMP}} P(\frac{B}{1})_{\text{SIMP}} P(\frac{\text{IN}}{2})_{\text{DUP}} \right\} \times \\
 &\quad \left\{ P(\frac{\text{AIL}}{1,2})_{\text{DUP}} P(\frac{\text{RUD}}{1,2})_{\text{DUP}} P(\frac{\text{ELEV}}{1,2})_{\text{DUP}} P(\frac{\text{OUT}}{1})_{\text{DUP}} + P(\frac{\text{AIL}}{1})_{\text{SIMP}} P(\frac{\text{RUD}}{1})_{\text{SIMP}} P(\frac{\text{ELEV}}{1})_{\text{SIMP}} P(\frac{\text{OUT}}{2})_{\text{DUP}} \right\} P(\frac{\text{DCU}}{1})_{\text{DUP}} \\
 &\quad + \left[P(\frac{A}{1})_{\text{SIMP}} P(\frac{B}{1})_{\text{SIMP}} P(\frac{\text{IN}}{1})_{\text{SIMP}} P(\frac{\text{OUT}}{1})_{\text{SIMP}} P(\frac{\text{AIL}}{1})_{\text{SIMP}} P(\frac{\text{RUD}}{1})_{\text{SIMP}} P(\frac{\text{ELEV}}{1})_{\text{SIMP}} \right] P(\frac{\text{DCU}}{2})_{\text{DUP}} \quad (27)
 \end{aligned}$$

The various reliability expressions with appropriate failure rates and exposure times may be substituted into each state's probability factor in equation (27) to obtain a numerical result.

2.4 Effect of Failure Coverage

Failure coverage is defined as the probability of successfully detecting a failure within a redundant stage, isolating that failure to the specific LRU, and reconfiguring the stage to place the failed LRU off-line. It is generally accepted that coverage values of unity are possible with 3 or more healthy redundant LRU's. The problem arises when a failure occurs when just previous there were only 2 healthy LRU's-which has failed? Therefore, the coverage factor (c) is used to modify the probability of achieving the simplex state.

The Stage-State method includes coverage by splitting the simplex state for a stage into two parts-that which successfully degrades to simplex, and that which does not. This is now illustrated for a duplex stage. Equation (2) is rewritten below for a duplex stage with identical LRU's.

$$P(\text{good or bad}) = 1.0 = \underbrace{R^2}_{ST1} + \underbrace{2RQ}_{ST2} + \underbrace{Q^2}_{ST3} \quad (28)$$

If success is defined as having at least 1 of the 2 LRU's healthy, then the success states include states 1 and 2 only. State 2 represents the 2 ways in which simplex operation can be achieved. When coverage is not unity, the probability of achieving this state is $2RQc$. In this process a new unsuccessful state has evolved, namely $2RQ(1-c)$. Now

$$P(\text{good or bad}) = 1.0 = \underbrace{R^2 + 2RQc}_{\text{Success States}} + \overbrace{2RQ(1-c) + Q^2}^{\text{previously 1 state Failure States}} \quad (29)$$

A triplex stage can be evaluated in a similar fashion. With unity coverage the triplex states are shown by equation (30).

$$P(\text{good or bad}) = 1.0 = \underbrace{R^3}_{ST1} + \underbrace{3R^2Q}_{ST2} + \underbrace{3RQ^2}_{ST3} + \underbrace{Q^3}_{ST4} \quad (30)$$

If success is again defined as successfully achieving at least simplex operation, then the first 3 states represent success. State 3, however, must be modified by c if the coverage is not unity so that

$$P(\text{good or bad}) = 1.0 = \underbrace{R^3 + 3R^2Q + 3RQ^2c}_{\text{Success States}} + \overbrace{3RQ^2(1-c) + Q^3}^{\text{previously 1 state Failure States}} \quad (31)$$

3.0 Conclusion

Traditional reliability analysis methods are error-prone and difficult to use for complex flight control systems possessing a large number of LRU types and the redundancy management of individual LRU's. This is primarily due to the large number of combinations of possible success states and dependencies. The Stage-State reliability analysis method is a much simpler approach which is well-suited to use by the system architect. The method makes it readily apparent where the sources of the system unreliabilities are located. Also, because of its simplicity, fewer errors arise and the use of small portable computers is possible.

4.0 References

- 1) Rice, J.W., 20 Dec 1979, Digital Flight Control Reliability - Effects of Redundancy Level, Architecture, and Redundancy Management Technique, Seattle, Washington, Boeing Document D180-25578-1.

Also published as technical paper without the Stage-State Reliability Analysis appendix as:

Rice, J.W. and McCorkle, R.D. August 1979. Digital Flight Control Reliability - Effects of Redundancy Level, Architecture, and Redundancy Management Technique. Boulder, Colorado. AIAA Guidance and Control Conference. AIAA paper #79-1893.

METHODOLOGY FOR MEASUREMENT OF FAULT LATENCY IN A DIGITAL

AVIONIC MINIPROCESSOR*

John G. McGough
Fred Sverrn
Flight Systems Division
Bendix Corporation
Teterboro, New Jersey 07608

and

Salvatore J. Bavuso
NASA Langley Research Center
Hampton, Virginia 23665

ABSTRACT

Using a gate-level emulation of a typical avionics miniprocessor, fault injection experiments were performed to (1) determine the time-to-detect a fault by comparison-monitoring, (2) forecast a program's ability to detect faults and (3) validate the fault detection coverage of a typical self-test program.

To estimate time-to-detect, six programs ranging in complexity from 6 to 147 instructions, were emulated. Each program was executed repetitively in the presence of a single stuck-at fault at a gate node or device pin. Detection was assumed to occur whenever the computed outputs differed from the corresponding outputs of the same program executed in a non-faulted processor. Histograms of faults detected versus number of repetitions to detection were tabulated.

Using a simple model of fault detection, which was based on an analogy with the selection of balls in an urn, distributions of time-to-detect were computed and compared with those obtained empirically.

A self-test program of 2,000 executable instructions was designed expressly for the study. The only requirement imposed on the design was that it should achieve 95% coverage. The program was executed in the presence of a single stuck-at fault at a gate node on device pin. The proportion of detected faults was tabulated.

In all experiments faults were selected at random over gate nodes or device pins.

1. INTRODUCTION

1.1 Background

NASA's Langley Research Center has been actively pursuing the synthesis of a reliability assessment capability for fault-tolerant computer-based systems for several years. This work has culminated in the development of CARE III (Computer-Aided Reliability Estimation) which is a general purpose reliability assessment tool for highly reliable fault-tolerant systems tailored toward flight critical avionic systems employing multiple digital computers. A major innovation of CARE III is its treatment of coverage which is a vital factor in the reliability modeling of digital fault-tolerant computer systems. Coverage, a generic term, captures the notion of a system's ability to handle hardware faults and involves system fault detection, isolation of the fault to a reconfigurable (redundant) hardware module, and fault reconfiguration and recovery. The first two components have been modeled extensively and have been shown to be critical for achieving high system reliabilities.

What is also evident in the literature is a lack of empirical coverage data although several very powerful reliability evaluators require this data. As a result, a pilot study was conducted in 1978 to test the feasibility of measuring detection coverage and investigating the dynamics of fault propagation in a digital computer. The specific objectives were to study how typical software causes stuck-at faults to propagate and hence become detectable, to account for as many software code characteristics (e.g., instruction subset, branching) as possible affecting detection (with an eye toward optimizing fault detection by code synthesis), and to determine a method of forecasting a given software program's detecting ability prior to computation. A series of fault injection experiments were conducted using a gate-level simulation of a small idealized processor with a limited instruction set. The results of the study were surprising since they contradicted the prevailing belief that most hardware faults cause catastrophic and hence detectable computational errors. In fact, a significant proportion of faults remained latent after many repetitions of a program. The ramifications of these observations can have a significant impact on the design of fault-tolerant digital computers which employ comparison-monitoring or majority-voting for fault detection and isolation. The risk is associated with the accumulation of latent and therefore undetected faults which may defeat the comparison-monitoring or majority-voting detection schemes. Needless to say, these considerations are of paramount importance to reliability assessment; as a result, NASA funded another study to investigate the findings of the pilot study as it was not clear from the pilot study that similar results could be obtained for a real processor—the follow-on work was based on a real avionic processor. This work was also extended to evaluate an airborne self-test program, to account for undetected faults, and to assess the significance of injecting faults at the gate-level and at the functional pin-level.

* The contents are based on the study:

"Methodology for Measurement of Fault Latency in a Digital Avionic Miniprocessor", NAS 1-15946, Flight Systems Division, Bendix Corporation, sponsored by Langley Research Center, Hampton, Va.

1.2 Objectives of the Study

A primary objective of the present study is to ascertain whether the results of the previous study apply to a real avionics processor. Specifically,

- Given a set of software programs ranging from a simple "fetch and store" to a complicated, multi-instruction algorithm inject a single fault, selected at random, and observe the time to detection. Detection is assumed to occur whenever there is a difference between the computed outputs of the faulted and non-faulted processors executing the same program. Determine differences in detection time when faults are injected at the gate-level and component-level.
- Based upon empirical distributions, develop and validate a model of fault latency that will forecast a program's fault detecting ability.

The following additional objective was added,

- Given a typical avionics self-test program inject faults at both the gate-level and component-level and determine the proportion of faults detected.

2. EMULATION DESCRIPTION

2.1 BDX-930 Architecture

The Bendix BDX-930 Digital Processor is a microprogrammed, pipelined machine designed around the AMD2901A four bit microprocessor slice. The machine contains sixteen general purpose registers of which four registers may be loaded directly from memory and two registers may be used as base registers. One register is used as a stack pointer.

The program counter and memory address register are contained in the 9407, a chip designed to perform memory address arithmetic. Along with a temporary register contained on the same chip, the BDX-930 is able to perform four basic addressing modes involving three registers and various instruction fields.

The machine contains three memory interface data registers which are used to input and output memory data. There are also a number of one bit status flag registers that can be manipulated under program control. This includes the F1 and F2 registers, which are hardware flags, and the interrupt enable, overflow status registers. There also exists the indirect and link registers used by the microcode for branching.

The microcode is contained in seven proms and a pipeline register is included for simultaneous microcode fetch and decoding. Various internal and external conditions can affect microcode branching as selected by the microcode itself and a microcode control prom. In addition to a rich instruction set which includes 16 and 32 bit fixed point operations, there is a test set interface in the microcode. A selectable saturate mode is available which limits the results of arithmetic operations when overflow or underflow occur.

For simulation purposes, the computer has been divided into six partitions:

1. Address Processor
2. Data and Status Registers
3. Microcontroller
 - Pipeline Register
4. ALU (2901A)
5. Microcode
6. Control Proms

The partitioning is roughly equivalent to the stages of the pipe: - address, fetch, decode, and execute. These stages of the pipe are joined by various buses throughout the CPU. These buses are formed from tri-state logic and some are bidirectional.

A list of the devices used in the BDX-930 and their failure rates is given in Table 1, obtained from MIL-HDBK217B, Notice 2.

2.2 Description of the Emulator

The emulation includes the components of the CPU (Central Processor Unit), scratchpad memory and those portions of the program memory containing six target programs and the target self-test program. The emulation is derived from the circuit schematics of the BDX-930 and includes all of the devices identified in those schematics. Each device is represented by a gate-level equivalent circuit supplied by the chip manufacturer. It was found that six types of gates were sufficient to represent any device, e.g., NAND, NAD, OR, NOT, NOR, EXCLUSIVE OR. Table 2 gives the number of equivalent gates in each device of the CPU. In all, 5,100 gates were required.

All devices of the CPU were represented at the gate-level except the following:

16 general purpose arithmetic registers

program memory

scratchpad memory

microprogram and control memories

which are represented at the functional level.

The emulation did not include the direct memory access unit (DMA) or any of the devices of the I/O. The emulated devices the CPU are shown in Figure 1.

Faults were injected into all devices except the program and scratchpad memories. Because the program is "read-only", no processor, faulted or not, is permitted to write into this memory. However, even though the scratchpad memory is never faulted, a faulty processor can write into it. As a consequence, in the parallel mode of operation where 36 processors are simultaneously emulated, the corresponding 36 scratchpad memories are also emulated.

No delay has been simulated between logic gates. It is assumed that all combinational logic is stable at the output the instant an input pattern is applied to it. This means that each time the input is changed, the network need only be evaluated once to supply the correct output pattern. Operating in this manner is very time efficient, but puts stringent requirements on the order of evaluation of the gates. To be able to meet these requirements, the logic is levelized, i.e., placed in groups or levels that represent the proper order of evaluation.

The emulator utilized the parallel method of logic simulation (see, for instance, (Seshu, S., et al 1962; Hardie, F.H., et al 1967)). The data word of a PDP-10 contains 36 bits; each bit position is used to represent a different machine. The simplest gate operations are represented by a single Boolean instruction; when the two inputs occupy the same bit positions in their respective words, the output also occupies this bit position. The advantage of this technique is execution time savings. Typically, the amount of code necessary to simulate 36 machines is of the same order as the amount of code necessary to simulate only one machine. For an additional increase in speed the BDX-930 description is contained in compiled code, rather than in tables.

Certain portions of the machine, notably the memory elements, were represented at a functional level rather than a gate level. For microprogram memory, two words of PDP-10 storage contain 56 bits of micro-store; at micro memory fetch time, these bits are retrieved from the proper address for each of the simulated machines and combined to form suitable words to interface the gate portion of the emulation. The ROM portion of macro memory is handled in the same manner. Writeable store contains a routine to translate the gate inputs into consecutive PDP-10 storage words so that there is one copy of writeable storage for each machine being emulated. On reading this storage, the process is reversed.

In a typical run of the emulator, 36 different machines are exercised; 35 faulted machines and one good machine. Each faulted machine is assumed to have a single solid fault at one node, either stuck-at-one (SA1) or stuck-at-zero (SA0). The faults are injected by defining extra gates at each node, an AND gate for stuck at zero and an OR gate for stuck at one. A typical AND gate using this technique is shown in Figure 2.

An additional reduction in run-time can be achieved by observing that not all gate faults are distinguishable at the gate output. For example, an SA0 fault on the input node of an AND gate is indistinguishable from an SA0 fault on the output node. As a consequence, if two or more indistinguishable faults of the same gate are selected, only one fault will be emulated.

It will be noted that only one partition of the BDX-930 runs with faults injected in each simulated run. The remaining partitions run 'true value', that is logic without fault injection capabilities. This results in a time saving in program execution. When the entire emulator is run true-value, the execution ratio between PDP-10 time and simulated time is 21,000:1, with faults injected in one partition, this number is approximately 25,000:1.

3. FAULT MODELLING AND SELECTION

3.1 Fault Model

In the present study the following assumptions are made regarding failure modes:

- Every device can be represented, from the standpoint of performance and failure modes, by the manufacturer-supplied, gate-level equivalent circuit.
- Every fault can be represented as either an S-a-0 or S-a-1 fault at a gate node.
- The failure rate of the device is equally distributed over the gates of the equivalent circuit.
- The failure rate of a gate is equally distributed over the nodes of the gate.
- S-a-0 and S-a-1 faults are equally likely.
- Memory faults are exclusively faults of single bits.
- A memory fault is the complement of its non-faulted state.

Faults are injected into all devices except the main memory. In the case of the microprogram memory, which is emulated at the functional level, faults are injected into the memory cells where they remain active for the duration of the test. Faults are injected at an input or output gate node, and also remain active for the duration of the test. When a fault is injected at an output node it is allowed to propagate to all nodes and devices that are physically connected to the failed node. When a fault is injected at an input node it does not propagate back to the driving node. This strategy provides a wider variety of failure modes than would otherwise be possible if propagation were allowed. The resultant fault set includes a rich assortment of static and dynamic (i.e., data-dependent) faults.

The above procedure does not distinguish between gate-level and component (i.e., pin)-level faults except by probability of occurrence; the method automatically assigns failure rates to pins. However, a different selection procedure was employed for component-level faults. For these faults it was assumed that the failure rate of each device is equally distributed over the pins.

While this assumption violates the prescribed fault model it is consistent with the conventional method of estimating fault detection coverage by simulating faults in actual hardware.

4. DESCRIPTION OF EXPERIMENTS

4.1 Definition of Failure Detection

In the present study fault coverage and latency estimates are obtained by employing two, conventional techniques of failure detection: comparison-monitoring and self-test.

In comparison-monitoring a set of computed variables is compared with a corresponding set computed in another processor. If it is arranged that both processors operate on identical inputs and are closely synchronized, then any difference in a computed variable signifies that one of the processors has failed. In practice each processor executes an algorithm which compares the appropriate variables and signals a discrepancy when such exists. In the present study this algorithm was omitted; a fault is considered to be detected if a difference between corresponding variables exists irrespective of the ability of either processor to recognize the difference or signal the discrepancy. Thus, the fault coverage obtained from the study is somewhat more optimistic than would be obtained in practice.

In self-test, on the other hand, each component of the processor is exercised by a set of computations designed specifically to test that component. The results of each computational set are compared with pre-stored values and any difference signifies that the fault was detected. In practice, and in the study, the processor increments a register after the successful completion of each test and before proceeding to the next test. If the test is not successful the program exits. After an interval of time equal to the maximum time to complete the program, the contents of the counter are decoded. If the value exactly equals the total number of tests, the fault was not detected. Otherwise the fault was detected.

4.2 Definition of Failure Detection Coverage

We assume that a test procedure is given for detecting failures of a component, C . Each failure mode of C will require a non-zero time for detection. By considering all failures of C and all combinations of inputs and internal states of C , we obtain in principle, if not in practice, a probability density function for time-to-detect, which is measured from the onset of the failure to the time of detection. Denoting this density by $\text{pdf}(\tau)$ where

$$\tau = \text{time-to-detect} = \text{latency time}$$

we define

Test Coverage

$$1) \quad 1 - \alpha(\tau) = \int_0^{\tau} \text{pdf}(x) dx$$

= probability of detecting a failure of C in the interval $0 \leq t \leq \tau$.

Observe that, according to this definition, test coverage is a function of latency time. The definition can be extended to all devices of the computer as follows:

Subdivide the computer into mutually exclusive components C_1, C_2, \dots, C_k with failure rates $\lambda_1, \lambda_2, \dots, \lambda_k$, and test coverages $1 - \alpha_1(\tau), 1 - \alpha_2(\tau), \dots, 1 - \alpha_k(\tau)$, respectively.

Set $\text{pdf}_i(\tau)$ = probability density for time-to-detect failures of

$$C_i, i = 1, 2, \dots, k.$$

Then the pdf for all failures of the computer is

$$2) \quad \text{pdf}(\tau) = \sum_{i=1}^{i=k} \frac{\lambda_i}{\lambda} \text{pdf}_i(\tau)$$

where

$$\lambda = \lambda_1 + \lambda_2 + \dots + \lambda_k.$$

Test coverage of the whole computer is then

$$3) \quad 1 - \alpha(\tau) = \sum_{i=1}^{i=k} \frac{\lambda_i}{\lambda} (1 - \alpha_i(\tau)).$$

From 3 we obtain

$$4) \quad \alpha(\tau) = \sum_{i=1}^{i=k} \frac{\lambda_i}{\lambda} \alpha_i(\tau), \text{ as expected}$$

One of the objectives of the present study is to obtain estimates of the probability density function, pdf(-). These estimates are presented in Section 7.

4.3 Indistinguishable Faults and Effects on Coverage

During the development of the emulator it became apparent that a significant proportion of components had no affect whatsoever on the digital process. For the most part, these components are associated with unused pins, e.g., a complementary output of a flip-flop. However, there are other components whose lack of effect are not as obvious as, for example, a component that only affects the process when it is faulted. Certain micromemory bits are in this category. In order to distinguish between these categories of faults we are lead to the following informal definitions:

A fault that cannot be detected by any test sequence is indistinguishable. All other faults are distinguishable.

Effects on Coverage

The presence of indistinguishable faults can lead to erroneous and misleading estimates of coverage. In theory, indistinguishable faults should be disqualified from the emulation or from the fault selection process. This is consistent with the definition of coverage which implicitly assumes that all faults are distinguishable. Unfortunately, in order to disqualify indistinguishable faults from the emulation or from the fault selection process they must be first identified, a non-trivial task. The approach taken in this study was to select faults without regard to their distinguishability properties and analyze only those faults that were undetected by Self-Test. The proportion of indistinguishable faults from this set was then used as an estimate over all faults.

We now indicate, briefly, how indistinguishable faults affect coverage.

If

γ = proportion of components yielding indistinguishable faults

and

$1 - \alpha$ = coverage of distinguishable faults

then

$1 - \alpha$ = desired coverage

and

5) $(1 - \alpha)(1 - \gamma)$ = coverage when indistinguishable faults are counted as undetected. We note, incidentally that

6) $(1 - \alpha)(1 - \gamma) + \gamma$ = coverage when indistinguishable faults are counted as detected.

The estimate of (5) will be obtained if indistinguishable faults are not disqualified. Then, coverage estimates will be in error by the factor, $1 - \gamma$.

One of the objectives of the experiments is to estimate τ for a variety of computations including self-test. The Phase I experiments consist of six software programs ranging from a simple fetch and store to a complicated multi-instruction, linear convergence algorithm. Using comparison-monitoring the probability distribution for τ will be estimated for each of the six programs and the interdependence of these distributions and the number and type of instructions will be ascertained.

The Phase II experiments utilize a typical avionics system self-test program which consists of 241 separate, sequential tests. The program consists of 2000 executed instructions which requires an execution time of 3 milliseconds on the BDX-930.

4.5 Phase I Experiments

This phase consisted of six programs each of which was coded in the assembly language of the BDX-930. For the purpose of comparison with the experiments in (Nägel, P., 1978) the instructions of the BDX-930 were primarily restricted to the following set:

LOAD
STORE
ADD
SUBTRACT
BRANCH

In the following programs the initializing variables were stored simultaneously in the 36 copies of the scratchpad memories. The ground rules governing the experiments were:

- o After each repetition of the program (for a randomly selected set of inputs) by the non-faulted processor the output variables from each repetition are compared, term by term, with those of the faulted processors. The first repetition which resulted in a miscomparison is referred-to as the "time-to-detect" or "latency time".
- o At the start of each experiment the initial conditions for all subsequent repetitions were stored in successive locations of memory and the program counter was preset to the address of the first instruction.

An experiment consisted of executing one of the following programs in the presence of a single fault.

a) FIBONACCI (FIB)

Create a Fibonacci series starting with a pair of integers. Eight terms are generated, each term constituting a repetition.

b) FETCH AND STORE (FETSTO)

Fetch an integer from memory and store in another location. This process is repeated eight times.

c) ADD AND SUBTRACT (ADDSUB)

Fetch two integers from memory and compute the difference and sum and restore in memory. Repeat eight times.

d) SEARCH AND COMPUTE (SERCOM)

Fetch three integers, A, B, C, from memory and set

$$\begin{aligned} S_1 &= B + C \\ S_2 &= B && \text{if } B \leq A \\ S_1 &= B + C \\ S_2 &= B - C && \text{if } A < B \text{ and } C \leq A \\ S_1 &= B - C \\ S_2 &= BC && \text{if } A < B \text{ and } A < C \end{aligned}$$

and store S_1 and S_2 in memory. Multiplication is performed by successive addition.

e) LINEAR CONVERGENCE (LINCON)

A line, characterized by slope, M, and Y-intercept, Y, is given. A positive abscissa, X, is selected at random. By successively incrementing or decrementing the slope, M, by 1 the slope is adjusted to obtain a line that crosses the X-axis prior to X and has a minimum deviation from the X-axis. The new slope and ordinate at X are stored in memory for comparison.

f) QUADRATIC (QUAD)

Fetch four integers A, B, C, X, from memory and compute and store $AX^2 - BX - C$. Multiplication is performed by repeated addition. The process is repeated four times. The type and frequency of the instructions executed in the above programs are given in Table 3.

4.6 Phase II Experiments

This phase consists of injecting faults and executing a typical avionic flight control system self-test program to determine failure detection coverage. The self-test program was written expressly for this study.

The task of designing the self-test was given to an experienced programmer with considerable expertise in self-test. The only requirement imposed was that the resultant test should achieve a coverage of 95%. The result was a program consisting of 2000 executed instructions with an execution time of 3 milliseconds. The detection strategy was that of exercising every instruction type at least once and, in most cases, with numerous variations.

Self-Test

The program consisted of 241 subtests. After a successful completion of a test the program increments a register and proceeds to the next test in the sequence. If, however, a failure is detected the

program skips the remaining tests and transfers the contents of the register to a designated memory location whose contents became the measure of failure detection. In the Phase II experiments a fault was defined as detected if, after a complete execution of the self-test program by the non-failed processor, the contents of the designated memory did not equal 241 in the faulted processor. Observe that, according to this definition, a fault is detected if the faulted processor jumps out of the program, gets hung-up in an infinite loop or executes a single extra instruction before transferring the contents of the incremental register to memory.

5. URN MODEL

5.1 Urn Model Description

Several models have been investigated in an attempt to characterize the dynamics of fault propagation in a digital computer. Although simplistic in their assumptions, these models may, nevertheless, provide insight into this undoubtedly complex process. It has been conjectured (Nagel, P., 1978) that the distribution of latency can be modelled by analogy with balls in an urn. We prefer to employ a different analogy although the resultant distributions are the same.

We postulate that the computer can be subdivided into three sets of mutually exclusive components C_1, C_2, C_3 such that

C_1 = Set of components randomly exercised by the program

C_2 = Set of components continually exercised by the program

C_3 = Set of components never exercised by the program.

We make the further assumption that a fault is detected if and only if the faulted component is exercised. The scenario is that of an avionics computer executing two software programs one of which is executed full-time and the other, part-time. The components that are exercised by the full-time mode are denoted by C_2 and those exercised by the part-time mode by C_1 . Neither the full-time or part-time modes exercise components, C_3 .

We assume that the part-time mode is exercised randomly. If the unit of time is a repetition of the full-time program then we postulate that the excitation is poisson-distributed in time with

a = probability that the part-time mode is exercised in a repetition of the full-time program.

Let λ_1 = Failure rate of C_1 (Failures/hour)

λ_2 = Failure rate of C_2 (Failures/hour)

λ_3 = Failure rate of C_3 (Failures/hour)

$\lambda = \lambda_1 + \lambda_2 + \lambda_3$ (Failures/hour)

We now derive the latency distribution given that a fault has just occurred. The distribution is defined in terms of three parameters, a , P and Q_0 where

P = Probability that the fault is detected in the first repetition given that it occurred in sets C_1 or C_2

Q_0 = Probability that the fault is never detected.

It is easy to derive the following relationships:

$$7) \quad P_0 = 1 - Q_0 = \frac{\lambda_1}{\lambda} + \frac{\lambda_2}{\lambda}, \quad Q_0 = \frac{\lambda_3}{\lambda}$$

$$8) \quad P = \frac{\frac{\lambda_2}{\lambda} + a \frac{\lambda_1}{\lambda}}{\frac{\lambda_2}{\lambda} + \frac{\lambda_1}{\lambda}} = \frac{\frac{\lambda_2}{\lambda} + a \frac{\lambda_1}{\lambda}}{P_0}$$

If

P_k = probability that the fault is detected in the k -th repetition and not detected in a previous repetition, $k = 1, 2, 3, \dots, n$

q_{n+1} = Probability that the fault is not detected in the previous n repetitions

then

$$\begin{aligned}
 p_1 &= p_0 P = \frac{\lambda_2}{\lambda} + a \frac{\lambda_1}{\lambda} \\
 p_2 &= (1 - P) a p_0 = a (1 - a) \frac{\lambda_1}{\lambda} \\
 &\vdots \\
 9) \quad p_n &= (1 - P) (1 - a)^{n-2} a p_0 = a (1 - a)^{n-1} \frac{\lambda_1}{\lambda}, \quad n = 2, 3, \dots \\
 q_{n+1} &= q_0 + \sum_{k=n+1}^{\infty} p_k = q_0 + (1 - P) p_0 (1 - a)^{n-1} \\
 &= \frac{\lambda_3}{\lambda} + (1 - a)^n \frac{\lambda_1}{\lambda}, \quad n = 1, 2, 3, \dots
 \end{aligned}$$

Observe that

$$q_n + \sum_{k=1}^n p_k = 1, \text{ as expected.}$$

In estimating the above distribution the number of repetitions will be limited to eight. Then, the study will estimate the quantities

$$p_1, p_2, \dots, p_8, q_9$$

for S-a-1, S-a-0 and combined faults.

6. STATISTICAL ANALYSES

6.1 Estimators for Self-Test Coverage

The estimators for x , y and z are

$$10) \quad x^* = \frac{m_d}{m}$$

$$11) \quad y^* = \frac{n_d}{n}$$

$$12) \quad z^* = \frac{m_d + n_d}{m + n}$$

where

$x(y, z)$ = probability that a S-a-0 (S-a-1, combined) fault is detected;

$m_d (n_d)$ = number of S-a-0 (S-a-1) faults detected;

$m(n)$ = number of S-a-0 (S-a-1) faults injected.

6.2 Estimators for Latency

The estimators for x_k , y_k and z_k are

$$x_k^* = \frac{m_k}{m}$$

$$13) \quad y_k^* = \frac{n_k}{n}$$

$$z_k^* = \frac{m_k + n_k}{m + n}, \quad k = 1, 2, 3, \dots, 8,$$

where

$x_k(y_k, z_k)$ = probability that a S-a-0 (S-a-1 combined) fault is detected in the k -th repetition;

$m_k (n_k)$ = number of S-a-0 (S-a-1) faults detected in the k -th repetition.

With some abuse of terminology we define

$x_g(y_g, z_g)$ = probability that a S-a-0 (S-a-1, combined) fault is not detected in the previous 8 repetitions.

The estimators for x_g , y_g and z_g are

$$x_g^* = \frac{m - m_1 - m_2 - \dots - m_8}{m} = 1 - x_1^* - x_2^* - \dots - x_8^*$$

$$14) \quad y_g^* = \frac{n - n_1 - n_2 - \dots - n_8}{n} = 1 - y_1^* - y_2^* - \dots - y_8^*$$

$$z_g^* = \frac{m x_g^* + n y_g^*}{m + n} = 1 - z_1^* - z_2^* - \dots - z_8^*.$$

6.3 Estimators for Urn Model Parameters

The method of estimation will be described for S-a-0 latency distributions. With an obvious change in parameters, e.g. m_k , the estimates can be applied to S-a-1 and combined latency distributions, as well.

The method is based on the principle of maximum likelihood. We note that m_k S-a-0 faults are detected in the k -th repetition. Accordingly, we seek Urn Model parameters a , P and P_0 that maximize the likelihood function.

$$L = p_1^{m_1} p_2^{m_2} \dots p_8^{m_8} q_9^{m_9}$$

where

$$p_1 = P_0 P$$

$$p_2 = (1 - P) a P_0$$

$$15) \quad p_3 = (1 - P) a P_0 (1 - a)$$

.

.

$$p_8 = (1 - P) a P_0 (1 - a)^6$$

$$q_9 = P_0 + (1 - P) P_0 (1 - a)^7$$

and $m_9 = m - m_1 - m_2 - \dots - m_8$.

We note that q_9 corresponds to x_g of Section 6.2.

The maximum likelihood estimators for a , P and P_0 are obtained as the solution of

$$\frac{\partial L}{\partial a} = 0, \frac{\partial L}{\partial P} = 0, \frac{\partial L}{\partial P_0} = 0.$$

6.4 Accuracy and Confidence of Coverage Estimates

It can be shown (McFarlane, M. A., 1950) that

$$16) \quad E(x^*) = x, E(y^*) = y, E(z^*) = z$$

and

$$E((x - x^*)^2) = \frac{x(1-x)}{m}$$

$$17) \quad E((y - y^*)^2) = \frac{y(1-y)}{n}$$

$$E((z - z^*)^2) = \frac{z(1-z)}{m+n}$$

where

$$E(\cdot) = \text{expected value of } (\cdot).$$

For m , n sufficiently large the estimators x^* , y^* and z^* are approximately Gaussian with means and variances given by (16) and (17), respectively.

The following derivation of accuracy and confidence is general and applies to any quantity, x , estimated by the method of Section 6.2. As before,

x^* = estimate of x

m = sample size.

It is well-known (see ref. 3) that the probability that x lies between the limits

$$\frac{m}{m + \lambda^2} \left(x^* + \frac{\lambda^2}{2m} \pm \lambda \sqrt{\frac{x^* (1 - x^*)}{m} + \frac{\lambda^2}{4m^2}} \right)$$

or, equivalently, that x^* lies between the limits

$$(18) \quad x \pm \lambda \sqrt{\frac{x(1-x)}{m}}$$

is equal to λ , where λ is the area of the standard Gaussian distribution between $-\lambda$ and λ . From (18) we may say that the error in the estimate, x^* , is

$$(19) \quad \epsilon = \lambda \sqrt{\frac{x(1-x)}{m}}$$

with a confidence level of γ .

Equation (19) is an ellipse in x . Table 4 gives a tabulation of $\epsilon\sqrt{m}$ versus x for a confidence level of $\gamma = .95$.

It is often convenient to obtain error estimates that are independent of x . From (19) it can be seen that the maximum error occurs when $x = 1/2$. Table 5 gives a tabulation of this maximum error versus sample size and confidence level. It is noted that the maximum error can be extremely conservative.

7 RESULTS OF EXPERIMENTS

7.1 Distribution of Faults

Initially, 1,000 gate-level and 400 component-level faults were randomly selected. Later, in order to reduce the cost of the runs it was necessary to reduce the number of faults actually injected. The number of faults finally selected for each experiment are given in Table 6.

7.2 Phase I Experiments

The results of the Phase I experiments are summarized in Tables 7 and 8.

Table 7

This table shows the breakdown of faults injected versus faults detected in each of the six programs. Also shown is the percentage of undetected faults after completion of the specified number of repetitions of each program.

Table 8

This table gives the maximum likelihood estimates of a , P and P_0 , as defined in Section 7. Also shown are the resultant, computed, Urn Model distribution in terms of the occupancy probabilities of cells, 1, 2, ..., 8. These correspond to the probabilities x_i , y_i or z_i for S-a-0, S-a-1 and combined faults, respectively. In keeping with our previous notation, the occupancy probability of cell 9 is actually the probability that the fault is undetected in the previous 8 repetitions. As a comparison, the corresponding empirical distributions are also given.

Figure 3a through 5b show histograms of detected faults versus repetitions to detection for combined (i.e., S-a-0 and S-a-1) faults at both the gate and component-levels. Superimposed on each histogram is the distribution of the corresponding Urn Model.

7.3 Phase II Experiments

Indistinguishable Fault Estimates

In order to obtain an estimate of the proportion of indistinguishable faults each resultant, undetected fault was analyzed and those faults which were obviously indistinguishable were disqualified. At the gate-level, 71 out of 300 faults were identified as indistinguishable and, at the component-level, 11 out of 200 were identified as indistinguishable. Thus, the estimated proportion of components yielding indistinguishable faults are:

$$\gamma^* = \frac{71}{300} = 0.2366 \text{ at the gate level}$$

and $\gamma^* = \frac{11}{200} = .055 \text{ at the component-level}$

Since indistinguishable faults were not disqualified in the Phase I experiments all coverage estimates of Phase I should be multiplied by the appropriate $1 - \gamma^*$ factor, as prescribed in Section 7.

Self-Test Coverage

After disqualifying 71 indistinguishable faults 229 faults were effectively injected at the gate-level and 189 at the component-level. The resultant raw data is given in Table 9 by partitions.

As indicated previously, after each injected fault the self-test program was executed. Faults were generally detected either because an explicit test detected the fault or the fault caused a jump out of the program. These latter faults are denoted in Table 9 by "wild branches".

Summary of Results of Phase II Experiments

Gate-Level Faults

- 198 out of 229 combined faults were detected for a coverage of 86.46%.
- 100 out of 114 S-a-1 faults were detected for a coverage of 87.72%.
- 98 out of 115 S-a-0 faults were detected for a coverage of 85.22%.
- 9 out of 17 faults in Partition #5 were detected for a coverage of 52.94%.
- 5 out of 8 faults in Partition #6 were detected for a coverage of 62.5%.
- If faults in Partitions #5 and #6 are disqualified then 184 out of 204 faults were detected for a coverage of 90.2%.
- 103 out of the 198 faults detected resulted in wild branches, i.e., 52%.
- 95 faults were detected by an explicit test (even though it was not always possible to identify the test)
- Out of the 241 possible tests, at most 46 actually resulted in a detection, i.e., most of the tests were, effectively, redundant.

Component-Level Faults

- 185 out of 189 combined faults were detected for a coverage of 97.9%.
- 97 out of 100 S-a-1 faults were detected for a coverage of 97%.
- 88 out of 89 S-a-0 faults were detected for a coverage of 98.9%.
- 106 out of 189 faults detected resulted in wild branches, i.e., 56%.
- 79 faults were detected by an explicit test (even though it was not always possible to identify the test)
- Out of 241 possible tests, at most 44 actually resulted in a detection.

8. SUMMARY OF RESULTS OF EXPERIMENTS

8.1 Phase I Experiments

- Most detected faults are detected in the first repetition. Subsequent repetitions do not appreciably increase the proportion of detected faults.
- S-a-1 faults are easier to detect than S-a-0 faults.
- The micromemory contains a large proportion of indistinguishable faults
- A large proportion of faults remain undetected after as many as 8 repetitions
- Component-level faults are easier to detect than gate-level faults
- The coverage estimates of the Phase I experiments are not corrected for indistinguishable fault content.

Subsequent analysis of undetected faults indicates that the proportion of indistinguishable faults at the gate-level is 23.66% and 5.5% at the component-level. The combined, S-a-1 and S-a-0 coverage estimates should be corrected by dividing the raw coverage by $1 - \gamma^*$ where

$$1 - \gamma^* = .7633 \text{ for gate-level coverage}$$

$$= .945 \text{ for component-level coverage.}$$

The poor detection coverage of the six programs of Phase I is not surprising particularly if one considers that Self-Test, which exercises a much greater mix and quantity of instructions, achieves 86.5% detection (at the gate-level). Table 10 shows the instruction mix and quantity of instructions executed versus coverage for each of the six programs. By contrast, Self-Test exercises almost the entire instruction set of the CPU and executes approximately 2000 instructions in a single pass.

8.2 Phase II Experiments

- There is a significant difference in coverage of gate-level versus component-level faults, e.g., after disqualifying indistinguishable faults gate-level fault coverage was 86.5% whereas component-level fault coverage was 97.9%.
- There was a large proportion of indistinguishable faults in the gate-level emulation, e.g., 23.7%. The worst offender was the micromemory which yielded 33 indistinguishable fault out of a total of 41 selected.
- Only 48% of all detected faults were detected by an explicit test, i.e., 95 out of 198. 103 faults were detected because the fault resulted in a wild branch, i.e., a jump out of the first test.
- Most of the 241 tests comprising Self-Test were redundant; only 46 tests resulted in a detection.
- Of the 95 faults detected by an explicit test 59 were detected by the first 23 tests.
- This particular Self-Test was designed to exercise an instruction set rather than explicit hardware. As noted in Section 10, this approach results in an inefficient Self-Test since, it turned out, most of the tests exercised the same hardware.

8.3 Urn Model Distributions

From previous studies and results of experiments we make the following observations regarding the Urn Model.

- Despite its simplicity the Urn Model results in good correlation with all of the empirical distributions of the study. This is not surprising considering that the model has 3 degrees-of-freedom available for a best fit, i.e., P , P_0 and a , and the empirical distributions are heavily weighted in the first, second and last latency cells.

9. CONCLUSIONS

On the basis of the study we conclude:

- Emulation is a practicable approach to failure modes and effects analysis of a digital processor.
- The run time of the emulated processor on a PDP-10 host computer is only 20,000 to 25,000 times slower than the actual processor. As a consequence large numbers of faults can be studied at relatively little cost and in a timely manner.
- The fault model, although somewhat arbitrary, can be updated as more data becomes available.
- Gate-level equivalent circuits are available for digital devices including the 2901A.
- Gate-level faults are more difficult to detect than component-level faults.
- A computer self-test program of the order of 2000 executable instructions can detect 98% and possibly 99 or 100% of component-level faults. The feasibility of detecting the same proportions of gate-level faults remains to be determined.
- Emulation can be an important tool in the design of an efficient self-test.
- In a comparison-monitored system the accumulation of latent faults can be significant. In the study the proportion of undetected faults after 8 repetitions ranged from 40 to 62%.
- For the range of values considered the proportion of undetected faults after 8 repetitions is a linear function of the number of executable instructions.
- With a suitable choice of parameters the urn Model can be used to describe fault latency in a comparison-monitored system.
- Faults in the micromemory are difficult to detect.
- In a comparison-monitored system most detected faults are detected in the first repetition of the program. Subsequent repetitions do not appreciably increase the proportion of detected faults.
- A gate-level emulation of a real processor may contain a large proportion of indistinguishable faults. Identifying such faults is difficult.
- Only 48% of all detected faults were detected by an explicit subtest of Self-Test; 52% were detected because the fault resulted in a wild branch.

Concluding Remarks

The outcome of this study was no less surprising or intriguing than the results of the Nagel pilot study. Most of the data generated in the Nagel study were essentially duplicated in this study which in itself is remarkable because of the two "very different" hardware processors used in the studies (see Table 11 for a comparison). A significant finding of this work which correlates well with Nagel's observations is that comparison-monitoring yields a detection coverage which ranges from 40 to 60 percent and is in sharp contrast to assumed values of unity for first failure coverage in comparison-monitoring majority-voting (cm/mv) systems. Admittedly, the presence of undetected faults does not of and in itself constitute computer failure, but it does cast doubt on the validity of state-of-the-art reliability assessments and causes one to wonder what those latent faults are doing in the computer.

Another important finding of this study relates to the question of where faults should be induced, at the gate or pin (functional) level, to evaluate self-test computer programs. The study shows a wide dispersion in results between the two methods, i.e., 87-percent gate level versus 98-percent pin level. The issue is far from trivial because the proponent of pin-level testing can argue that the 11 percent that did not get detected by the pin-level method are don't-care faults anyway, whereas the proponent for the gate-level method argues that 11 percent of the faults are still present in the computer and may be manifested at a most inopportune time.

Currently it is simply not clear what impact latent faults could have in digital computers and their possible effects on fault-tolerant computer fault detection. The reliability analyst must be conservative when he cannot be accurate, so these findings must have a negative impact on reliability predictions for cm/mv systems, i.e., detection values based on gate-level fault injection must be used in reliability predictions in lieu of the pin-level value. Furthermore, these results strongly suggest a more conservative approach to fault detection in fault-tolerant systems utilizing cm/mv detection schemes. One approach to enhance detection would be to employ both cm/mv detection and concurrent periodic self-test.

Finally, the vehicle and its application which made these results possible deserve special emphasis. Although gate-level simulation is not new, the approach used in this study makes practical the generation of coverage data particularly for cm/mv schemes and opens up a new horizon of uses for such a tool, some of which were explored and reported on. Its use for designing efficient self-test code, identification of indistinguishable faults, a practical approach to failure modes and effects analysis, and fault analysis in general are just some that come to mind. In summary, gate-level simulation most assuredly will become an essential tool to design and reliability engineers.

10. RECOMMENDATIONS FOR FUTURE STUDIES

- The Phase I experiments should be repeated using flight critical, flight control computations. The instruction set should not be limited as it was in the present study. Additional tasks would include
 - Determination of the proportion of faults that affect the control surfaces.
 - Determination of the proportion of faults that prevent failure detection in the faulted processor.
- Investigate other methods of fault detection such as the use of redundant computations in a non-redundant processor in a flight critical, flight control application.
- Investigate the feasibility of extending the emulation to I/O interface devices such as AD and DA converters, I/O controllers, etc.
- Generate more realistic fault models. Perhaps manufacturers could be prevailed upon to supply equivalent circuits that are more closely correlated with failure modes as well as with performance.
- Develop a more realistic Urn Model. The resultant model could be an important tool in reliability modelling of a redundant system.

REFERENCES

1. Nagel, P., "Modeling of a Latent Fault Detector in a Digital System," Vought Corporation, Nasa contract, NAS1-13500, August, 1978.
2. Seshu, S. and Freeman, D. N., "The Diagnosis of Asynchronous Sequential Switching System," IRE Transactions on Electronic Computers, Vol. EC-11 No. 4, August, 1962, pp. 459-465.
3. Hardie, F. H., and Suhocki, R. J., "Design and Use of Fault Simulation for Saturn Computer Design," IEEE Transactions on Electronic Computers, Vol. EC-16, No. 4, August, 1967, pp. 412-429.
4. Cramer, H., Mathematical Methods of Statistics, Princeton University Press: Princeton, 1958.
5. McFarlane Mood, A., Introduction to the Theory of Statistics, McGraw-Hill: New York, 1950.

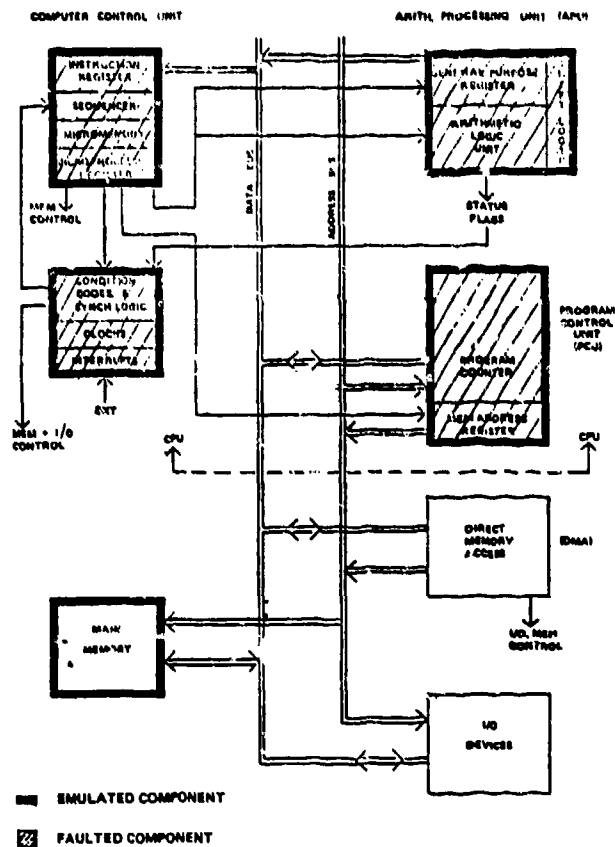


FIGURE 1 PROCESSOR ARCHITECTURE

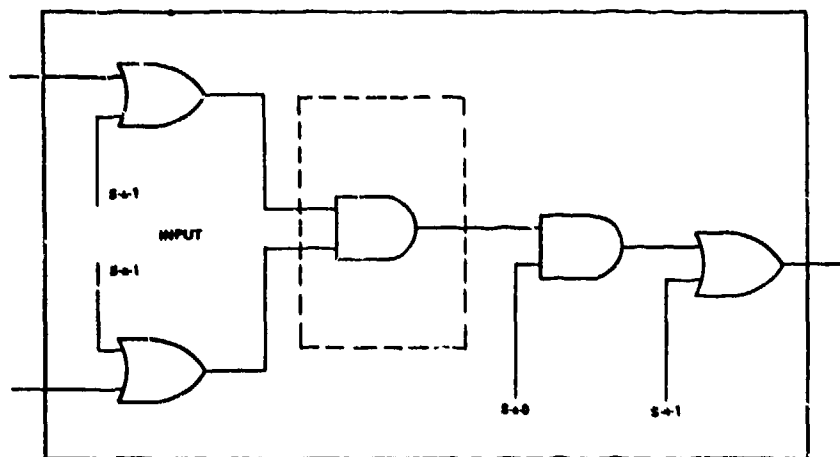


FIGURE 2 BASIC TWO INPUT AND GATE FAULT MODEL

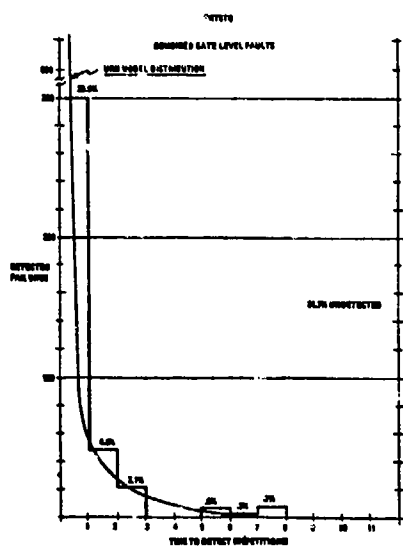


FIGURE 3A

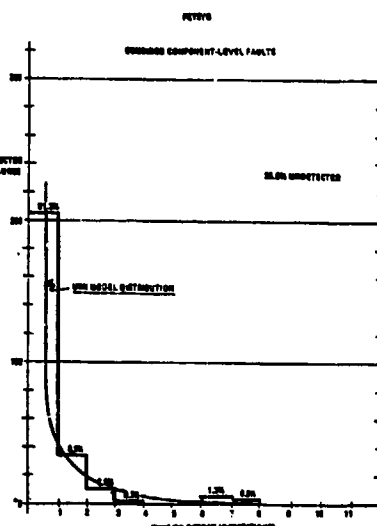


FIGURE 3B

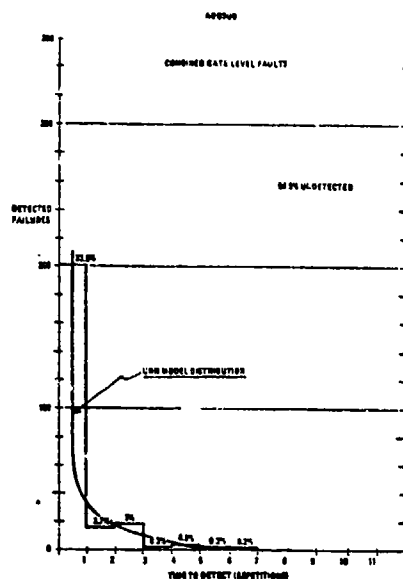


FIGURE 4A

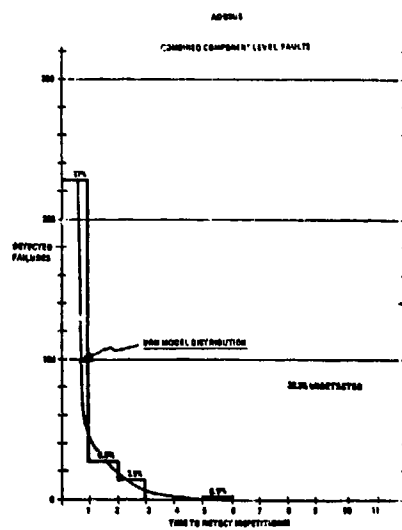


FIGURE 4B

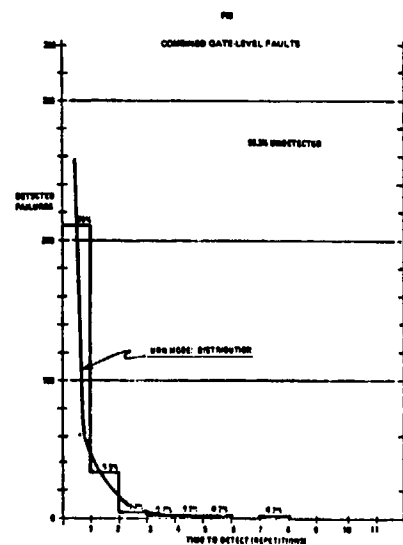


FIGURE 5A

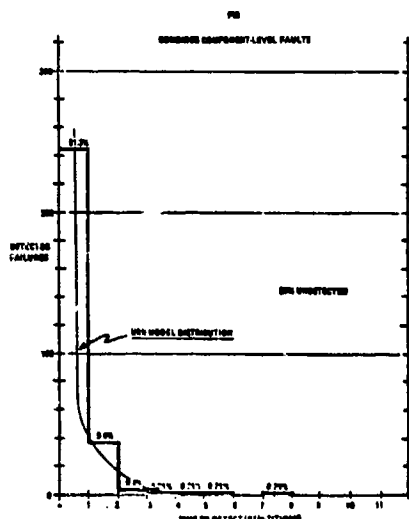


FIGURE 5B

TABLE 1 COMPONENTS OF THE BDR930 CPU

DEVICE	FAILURE RATE/PER UNIT (PPHM)
2901A	2.1656
2902	0.3898
5440	0.0653
54125	0.0855
54155	0.1448
54175	0.1755
54500	0.0855
54504	0.1073
54508	0.084
54510	0.0764
54520	0.0654
54532	0.2138
545151	0.1483
545288 (32x8 prom)	0.1787
545472 (512x8 proms)	1.009
54LS00	0.084
54LS02	0.074
54LS04	0.0943
54LS08	0.084
54LS11	0.0752
54LS32	0.084
54LS86	0.094
54LS113	0.1447
54LS151	0.1483
54LS193	0.1447
54LS198	0.1410
54LS169	0.6603
54LS175	0.1703
54LS245	0.3792
54LS253	0.1447
54LS257	0.1636
54LS273	0.6982
54LS283	0.2691
54LS352	0.3117
54LS367	0.1100
54LS374	0.7234
54LS377	0.7148

NOTE: The replacement for the 9407 includes the following devices:
54LS00, 54LS32, 54LS283, 54LS175, 54LS158 & 54LS257.

TABLE 2

MICROCIRCUITS AND EQUIVALENT GATE COUNT

DEVICE	EQUIVALENT GATES
2901A	798
2902	19
54113	8
54151	17
54153	16
54158	15
54169	58
54175	22
54245	18
54253	16
54273	34
54352	16
54374	26
54377	35
9407	143

TABLE 4

Error Ellipse for a Confidence Level of $\gamma = .96$

$$e/\sqrt{n} = \sqrt{x(1-x)}$$

TABLE 3 TYPE AND FREQUENCY OF INSTRUCTIONS EXECUTED

	FIB *	FETSTO	ADDSUB	SERCON	LINCON	QUAD
LOAD	2	1	2	5	18	7
STORE	1	2	2	6	28	5
ADD	3		2	17	16	30
SUBTRACT	0	1	1	1	4	1
BRANCH	4	2	2	24	39	38
TRANSFER	0		2	5	11	6
CLEAR	1				1	

TABLE 5

MAXIMUM ERROR VERSUS SAMPLE SIZE AND CONFIDENCE LEVEL

CONFIDENCE LEVEL \ SAMPLE SIZE	200	300	400	500	1000
.6	.03	.021	.021	.017	.013
.7	.037	.03	.026	.021	.017
.8	.046	.038	.033	.027	.021
.9	.058	.048	.041	.034	.026
.95	.069	.056	.049	.04	.031

TABLE 6
NUMBER OF FAULTS INJECTED

EXPERIMENT	GATE-LEVEL	COMPONENT-LEVEL
FETSTO	1000	400
ADDSUB	600	400
FIB	600	400
QUAD	863	400
SERCON	1063	400
LINCON	600	600
SELF-TEST	960	200

e/\sqrt{n}	x
0.0	0
.427	.06
.588	.1
.70	.15
.784	.2
.849	.25
.898	.3
.933	.36
.960	.4
.975	.45
.98	.5
.975	.55
.96	.6
.935	.66
.898	.7
.849	.75
.784	.8
.7	.86
.588	.9
.427	.96
0.0	1.0

DATE-LEVEL FAULTS										COMPONENT-LEVEL FAULTS											
PROGRAM IDENTIFICATION	INJECTED	S-a-B Faults			S-a-1 Faults			PERCENT UNDETECTED	PERCENT UNDETECTED	PERCENT UNDETECTED	PROGRAM IDENTIFICATION	INJECTED	S-a-B Faults			S-a-1 Faults			PERCENT UNDETECTED	PERCENT UNDETECTED	PERCENT UNDETECTED
		DETECTED	DETECTED FOR IDENTIFICATION	PERCENT UNDETECTED	INJECTED	DETECTED	DETECTED FOR IDENTIFICATION						PERCENT UNDETECTED	INJECTED	DETECTED	DETECTED FOR IDENTIFICATION	PERCENT UNDETECTED	INJECTED			
PETSTO (00)	503	136	27.0	63.3	497	183	36.8	59.8			PETSTO (00)	197	92	46.7	40.1	203	113	55.7	31.0		
ADDSUB (00)	296	84	28.4	34.2	304	117	38.5	54.9			ADDSUB (00)	197	100	50.8	37.1	203	128	63.1	27.6		
FIB (00)	296	90	30.4	42.2	304	120	39.5	54.3			FIB (00)	197	111	56.3	32.5	203	124	60.8	22.7		
SENCON (11)	503	183	36.4	62.6	497	212	42.7	57.3			SENCON (11)	197	114	58.9	43.1	203	147	72.4	27.6		
QMS (07)	296	114	38.5	57.4	304	166	47.8	49.3			QMS (07)	197	123	62.5	26.9	203	164	75.9	25.2		
LINCON (11)	218	148	68.0	30.0	304	182	59.9	46.7			LINCON (11)	197	146	74.1	28.9	203	166	70.8	21.2		

SUMMARY OF PHASE I RESULTS

TABLE 7

Exact estimate

Δ Empirical value

Occupancy probabilities

TEST		(1-a)	(a)	P	P ₀	C ₀₁₁ ¹	C ₀₁₁ ²	C ₀₁₁ ³	C ₀₁₁ ⁴	C ₀₁₁ ⁵	C ₀₁₁ ⁶	C ₀₁₁ ⁷	C ₀₁₁ ⁸	C ₀₁₁ ⁹
PETSTO/GATE	COMBINED T	0.5614	0.4386	0.7276	0.3049	0.299	0.001	0.021	0.002	0.007	0.004	0.002	0.001	0.617
	A					0.299	0.001	0.021	0	0	0.006	0.007	0.007	0.617
	S-a-B T	0.5168	0.4332	0.7313	0.3647	0.270	0.046	0.04	0.012	0.006	0.003	0.002	0.001	0.5162
	A					0.270	0.046	0.04	0	0	0.007	0	0.004	0.517
	S-a-1 T	0.6137	0.3863	0.6099	0.4049	0.323	0.030	0.011	0.011	0.007	0.004	0.002	0.002	0.5976
	A					0.328	0.052	0.004	0	0	0.004	0.005	0.010	0.599
PETSTO/COMP	COMBINED T	0.5760	0.4220	0.7927	0.0565	0.5125	0.003	0.033	0.018	0.009	0.005	0.003	0.001	0.3550
	A					0.513	0.005	0.033	0	0	0	0.013	0.008	0.355
	S-a-B T	0.6403	0.3397	0.7698	0.6068	0.467	0.047	0.031	0.021	0.014	0.009	0.006	0.004	0.4010
	A					0.467	0.061	0.036	0.005	0	0	0.025	0.005	0.401
	S-a-1 T	0.3594	0.6406	0.8070	0.6898	0.557	0.085	0.031	0.011	0.006	0.001	0.001	0.000	0.3103
	A					0.557	0.108	0.015	0	0	0	0	0.010	0.310
FIB/GATE	COMBINED T	0.3177	0.6823	0.8366	0.4184	0.350	0.047	0.015	0.005	0.001	0	0	0	0.5816
	A					0.350	0.055	0.007	0.002	0.002	0.002	0	0.002	0.582
	S-a-B T	0.2909	0.7091	0.8035	0.3784	0.304	0.053	0.015	0.004	0.001	0	0	0	0.6216
	A					0.304	0.064	0.003	0.003	0	0	0	0.003	0.622
	S-a-1 T	0.3466	0.6534	0.8632	0.4573	0.395	0.041	0.014	0.005	0.002	0.001	0	0	0.5427
	A					0.395	0.046	0.010	0	0.003	0.003	0	0	0.543
FIB/COMP	COMBINED T	0.2958	0.7042	0.8507	0.7200	0.613	0.076	0.022	0.007	0.002	0.001	0	0	0.280
	A					0.613	0.090	0.008	0.003	0.003	0.003	0	0.003	0.280
	S-a-B T	0	1.000	0.8473	0.6650	0.5635	0.1015	0	0	0	0	0	0	0.3357
	A					0.563	0.102	0	0	0	0	0	0	0.335
	S-a-1 T	0.4468	0.5532	0.8531	0.7738	0.660	0.063	0.028	0.013	0.006	0.003	0.001	0.001	0.2266
	A					0.660	0.079	0.015	0.005	0.005	0.005	0	0.005	0.227
ADDSUB/GATE	COMBINED T	0.5309	0.4691	0.8254	0.4058	0.335	0.033	0.018	0.009	0.005	0.003	0.001	0.001	0.5950
	A					0.335	0.027	0.030	0.003	0.005	0.003	0.002	0	0.595
	S-a-B T	0.6051	0.3949	0.7874	0.3604	0.284	0.030	0.018	0.011	0.007	0.004	0.002	0.001	0.6415
	A					0.284	0.024	0.036	0	0.007	0.007	0.003	0	0.642
	S-a-1 T	0.8353	0.5647	0.8536	0.4509	0.385	0.037	0.016	0.007	0.003	0.001	0.001	0	0.5493
	A					0.385	0.030	0.026	0.007	0.003	0	0	0	0.549
ADDSUB/COMP	COMBINED T	0.3401	0.6599	0.8413	0.6776	0.570	0.071	0.024	0.008	0.003	0.001	0	0	0.3225
	A					0.570	0.064	0.035	0	0	0.005	0	0	0.322
	S-a-B T	0.3133	0.6867	0.8064	0.8295	0.508	0.083	0.028	0.008	0.003	0.001	0	0	0.3705
	A					0.508	0.081	0.036	0	0	0.005	0	0	0.371
	S-a-1 T	0.3693	0.6307	0.8706	0.7242	0.631	0.059	0.027	0.008	0.003	0.001	0	0	0.2759
	A					0.631	0.054	0.034	0	0	0.005	0	0	0.276

UNR MODEL DISTRIBUTIONS
AND PARAMETER ESTIMATES

TABLE 8

UNR MODEL DISTRIBUTIONS
AND PARAMETER ESTIMATES

TABLE 9

GATE-LEVEL FAULTS (86.4% DETECTION)

PARTITION	DETECTED FAULTS		FAULTS INJECTED		DETECTED FAULTS		TOTAL DETECTED	TOTAL INJECTED
	m ₁	n ₁	m	n	TEST KNOWN	WILD BRANCH		
P1	14	14	16	16	1	27	28	32
P2	26	32	30	35	25	33	58	65
P3	20	19	20	19	17	22	39	39
P4	33	26	37	31	43	16	59	68
P5	2	7	8	9	7	2	9	17
P6	3	2	4	4	2	3	5	8
TOTAL	98	100	115	114	95	103	198	229*

m = S-a-0 faults

n = S-a-1 faults

* 71 faults were disqualified as indistinguishable

(COMPONENT-LEVEL FAULTS (97.7% DETECTION))

PARTITION	DETECTED FAULTS		FAULTS INJECTED		DETECTED FAULTS		TOTAL DETECTED	TOTAL INJECTED
	m ₁	n ₁	m	n	TEST KNOWN	WILD BRANCH		
P1	15	19	15	20	5	29	34	35
P2	38	34	38	35	35	37	72	73
P3	20	21	21	22	16	25	41	43
P4	15	23	15	23	23	15	38	38
P5								
P6								
TOTAL	88	97	89	100	79	106	185	189*

* 11 faults were disqualified as indistinguishable.

SELF-TEST DATA

TABLE 9

PROGRAM	TOTAL # OF EXECUTED INSTRUCTIONS	TYPE OF INSTRUCTION					GATE-LEVEL		COMPONENT LEVEL	
		LOAD AND STORE	ADD AND SUBTRACT	BRANCH	TRANSFER	CLEAR	PERCENT DETECTED 1st REPETITION	PERCENT UNDETECTED	PERCENT DETECTED 1st REPETITION	PERCENT UNDETECTED
FETSTO	6	3	1	2	0	0	29.9	61.7	51.3	35.5
ADDSUB	11	4	3	2	2	0	33.5	59.5	57.0	32.3
FIB	11	3	3	4	0	1	35.0	58.2	61.3	28.0
QUAD	87	12	31	38	6	0	43.2	53.3	71.8	23.5
SERCOM	59	12	18	24	5	0	39.5	60.5	64.8	35.3
LINCON	147	76	20	39	11	1	51.7	48.3	76.5	23.5

Note: This table is based upon one pass through the main program.

INSTRUCTION MIX versus DETECTION in PHASE I EXPERIMENTS

Table 10

REPETITION	FETSTO DETECTED		FIB DETECTED		ADDSUB DETECTED	
	REF. (1)	PHASE I	REF. (1)	PHASE I	REF. (1)	PHASE I
1	.187	.3	.261	.35	.313	.336
2	.051	.048	.057	.055	.096	.027
3	.017	.021	.047	.007	.067	.03
4	.017	0	.009	.002	.024	.003
5	.017	0	.028	.002	.014	.006
6	.042	.008	.033	.002	.014	.003
7	0	.002	.019	0	.019	.002
8	.025	.007	.009	.002	.004	0
9 (undetected)	.644	.817	.537	.582	.449	.898

TABLE 11

COMPARISON OF LATENCY ESTIMATES

Hierarchical Specification of the SIFT Fault Tolerant Flight Control System

P.M. Melliar-Smith and Richard L. Schwartz

Computer Science Laboratory
SRI International
Menlo Park, CA 94025

Abstract

This paper describes work in progress at SRI on the specification and mechanical verification of the Software Implemented Fault Tolerance (SIFT) flight control system. The methodology employed in the verification effort is discussed, and a description of the hierarchical models of the SIFT system is given.

Introduction

To meet the objectives of NASA for the reliability of safety-critical flight control systems, the SIFT computer must achieve a reliability well beyond the levels at which reliability can be actually measured. This paper describes the methodology employed to demonstrate rigorously that the SIFT computer meets its reliability requirements. We explain the hierarchy of design specifications from very abstract descriptions of system function down to the actual implementation. The most abstract design specifications can be used to verify that the system functions correctly and with the desired reliability, almost all details of the realization having been abstracted out. A succession of lower-level models refine these specifications to the level of the actual implementation, and can be used to demonstrate that the implementation has indeed the properties claimed of the abstract design specifications.

The SIFT (Software Implemented Fault Tolerance) computer is an aircraft flight control computer developed by SRI for the NASA ACHE program, under the direction of B. Dove and N. Murray of the Flight Electronics Division of NASA Langley Research Center. A SIFT system, designed to meet the required ultra high reliability by processor replication and voting, has been constructed by Bendix Corporation and is now operating at SRI. It will shortly be delivered to NASA Langley for evaluation in the AirLab. Rather than providing a general introduction to the SIFT system and the algorithms used to achieve the desired fault tolerance, we explore the process of refining the high level specifications of SIFT down to the implementation level. A general introduction to SIFT can be found in [5, 2] and a description of the SIFT executive appears in [4]. The SIFT hardware is documented in [1]. The fault-tolerance algorithms employed are defined in [2, 3].

Sections 1 and 2 of the paper present a brief introduction to the requirements of SIFT and the mechanisms employed to cope with the reliability requirements. Section 3 discusses how formal proof is used to substantiate the reliability claims. Section 4 outlines the specification hierarchy. Sections 5 through 8 describe each of the functional models in detail. The probabilistic analysis of system reliability is discussed in Section 9. Finally, Section 10 gives the current status of the project.

1. The Requirements for SIFT

The SIFT computer system has been designed to meet the requirements for future passenger aircraft control. Such aircraft must be designed to use significantly less fuel than current aircraft. Many design innovations are expected to assist in achieving the desired fuel economy, innovations in materials, structures, aerodynamics, engines, and almost every other aspect of aircraft design. Several of these innovations will require computer control of the flight of the aircraft, particularly to maintain the stability of the aircraft and to reduce the stresses in the structures of the aircraft. This computer control will be essential at all times to ensure the safety of flight. Existing aircraft use computers for various purposes, but never to perform flight safety critical functions, and thus do not have to meet the very demanding reliability requirements that apply to safety critical components of the aircraft.

The reliability requirement for a safety critical flight control computer, as proposed by FAA and NASA, allows a probability of life threatening failure no greater than 10^{-9} during a 10 hour flight. This is equivalent to a mean time between failures of about one million years of operation. The requirement allows higher rates for less critical failures, but the difficulty of assessing all the consequences of failures in computer systems has lead us to regard any deviation from the "correct" output as a failure of the system. The SIFT computer system has been designed not only to meet this reliability requirement but also to make it possible to demonstrate that this extreme requirement is indeed met.

2. The Role of Formal Proof

The extreme reliability requirement on SIFT imposes a very severe problem in substantiating the achievement of that level of reliability. At the required reliability rate, mere observation, even of a large number of systems, will be ineffective. Further, a SIFT system must be able to recover successfully from several million faults for every allowable system failure, and must therefore be able to recover from quite improbable and unforeseen faults and even combinations of faults. Thus validation by fault injection, while necessary, is unlikely to convince us that SIFT meets its reliability requirements.

The justification that SIFT meets the reliability requirement must be based on an extrapolation from fault rates that are easier to measure, such as those for an individual processor. For SIFT, this extrapolation takes the form of a discrete Markov analysis, with the numbers of working and faulty processors defining the states and the fault and reconfiguration rates defining the transitions. The validity of this extrapolation depends on a number of assumptions, and at the desired level of reliability, even 'minor' violations of the assumptions can have significant effects on the reliability achieved. Thus the assumptions must themselves be quite rigorously substantiated if the claimed reliability is to be believed. For instance, one important assumption of the Markov analysis is that the occurrence of faults is well described by a Poisson model with complete independence between processors. Much of the electronic and mechanical design of SIFT is intended to maintain this independence.

The validity of the Markov analysis depends also on the assumption that the states and the transitions of the Markov model correspond accurately to the actual system, and that the states in which system failure is possible are correctly identified. But this correspondence is far from obvious, for the actual system has very many states with many complex transitions between them, and the correspondence must be maintained even when one or more of the processors has suffered a fault. In SIFT, this correspondence is

The research reported herein was supported by the NASA Langley Research Center under Contract NAS1-15428.

based on a predicate system safe indicating that the replication of each of the tasks is sufficient so that the voting can mask the effects of the faults present in the system. The validation of SIFT now consists of two parts. The first of these is a demonstration that, so long as system safe is true, the system performs the desired flight control function, even though one or more processors may be faulty. This is a correctness property for the function performed by the system. The second is a demonstration that the Markov analysis computes an upper bound on the probability that system safe becomes false. This is a correctness property for the probabilistic reliability model of the system. Because even a very small defect in the demonstrations could allow failures at an unacceptable rate, these demonstrations must be performed with the rigor of mathematical proof.

The necessity for formal mathematical proof to ensure that SIFT meets the desired functional and reliability requirements presents two major issues:

- How does one define the criteria sufficient to ensure the correct functioning of the system?
- How does one prove that the criteria are satisfied by the actual system?

The first issue is crucial if the formal verification effort is to have any practical significance. One must have confidence, even as a non-computer scientist, that the formal specifications stating what is meant by the correct functioning of the system in fact reflect the intended behavior. That a formal specification expresses what the system designer intuitively means is determined by inspection. A formal specification must therefore be *believable* if rigorous mathematical correspondence to the specification is to ensure the desired effect. The larger and more complex the system, the more acute the problem becomes. Specifications reflecting the detailed behavior of the system allow the most straightforward formal verification effort but it is difficult to ensure that low-level specifications embody what is meant by the proper functioning of the system. Very high-level specifications, abstracting from the details of the system, are necessary if we are to state the overall functional and fault-tolerance properties of the system in a way that can be understood and believed. The problem then becomes one of reconciling the very high-level specifications with the detailed transformations performed by the programs of the actual system.

In order to state high-level system specifications that can be shown to be consistent with the actual program, one must formulate not just a single specification of the system, but a *hierarchy* of specifications. Our approach is to state a tiered set of models of the system, as illustrated in the following picture. Each model L_i in the hierarchy specifies an abstract view of the system, defining the properties of the system in terms of primitive predicates P_i and functions F_i employed at that level of abstraction. At each level in the hierarchy, a model L_i can be seen as a *refinement* of the previous level L_{i-1} . Correspondence between successive model levels is done by expressing each primitive function and predicate of higher-level L_i in terms of the functions and predicates of the lower-level L_{i-1} . With this mapping, one must then prove that each property derivable from the higher-level model can be proved from the lower-level model. By demonstrating this for all successive levels L_i and L_{i-1} , one can conclude by induction that any property provable from the highest-level model is also provable from the lowest-level model. Thus, the lowest-level model is consistent (or correct) with respect to the highest-level model, ensuring that analysis of the system based on a higher level model in the hierarchy is valid and could have been performed on the lowest-level model of the system.

Within the hierarchy, the lowest level model of the system is the actual program executed by the hardware, while the highest level model is chosen to allow the required properties of the system to be succinctly stated and analyzed. At different levels, models of the system are specified in different manners. At the most abstract level, a model of the system is defined by a set of *logical axioms*

L_1	(P_1, F_1)
L_i	(P_i, F_i)
L_{i+1}	(P_{i+1}, F_{i+1})
L_n	(P_n, F_n)

describing properties of the primitive functions and predicates. Such a model need not fully characterize the functional behavior of the primitives, instead specifying only the relevant properties. A *denotational*, or functional, model of the system can be used to provide a more concrete model of the system. The model is specified as a recursive function, providing an abstract implementation of the system. At this level of abstraction, the full functionality of the system is specified. A still lower-level is an *imperative* model of the system. Programs in a language such as Pascal or Ada are imperative models of a system, defining system function in terms of successive transformations of a global state.

Our hierarchy of models of the system makes use of each of the three types of models just discussed. We employ axiomatic models of the system at the highest levels of abstraction, denotational system models as intermediate levels of abstraction, and finally several levels of imperative models corresponding to the levels of software support involved in compiling and running SIFT.

As we mentioned above, verification of the hierarchy consists of demonstrating that each property derivable from a higher level model is supported by a lower level model. Between successive axiomatic models this is achieved by showing that, with the specified mappings, each axiom of the higher level model is provable as a theorem at the lower level model. Between an axiomatic model and a lower level denotational model, one must show that each function of the denotational model satisfies the axioms of the higher level model. Finally, in order to verify the relationship between a denotational model and an imperative model it is necessary to show that, based on a denotational model of the imperative language, the function performed by the imperative program is equivalent (homomorphic) to the function specified at the higher level denotational model.

3. An Outline of the Design of SIFT

The SIFT aircraft control computer system is designed to achieve high reliability from standard computers by replication of the hardware and adaptive voting implemented by software. The voting mechanism detects and masks hardware faults. Hardware detected to be faulty is reconfigured out of the system, with its workload being transferred to other processors. Thus several successive faults can be survived if there is sufficient time between them to permit the reconfiguration.

The system is constructed from up to eight identical computer units, each containing a Bendix BDX930 processor, a 32K main store, a broadcast interface, and a 1553 interface, as shown in Figure 1. The BDX930 is a 16 bit processor specifically designed for

military and aircraft use, with an instruction set reminiscent of, but not compatible with, Data General computers and a speed of rather less than one million instructions per second. Each BDX930 processor has its own 32k word main store, which cannot be accessed by any other processor. The 1553 interface provides a serial bus connecting the processor to the various aircraft sensors and actuators. The mean time between failures of one of these units, containing processor, store, and interfaces, is something less than one thousand hours.

The processors communicate with each other through the broadcast line, which contains the drivers and receivers for the star connected broadcast cables and a 324 word area of storage called the data file. The broadcast interface operates autonomously from the BDX930 processor, and is designed so that, if all processors broadcast simultaneously, the broadcast receivers will still be fast enough to receive and store all the information broadcast. The data file is divided into eight regions, one of which is used to hold information to be broadcast while the other seven regions are for the storage of information received from up to seven other processors. Thus, if a faulty processor broadcasts garbage, that garbage will all be placed in a specific region of every other processor's data file, where it can be ignored and where it cannot damage stored information being broadcast by other processors.

In SHFT there is conceptually a single instance of each logical task but, for reliability, that task is actually replicated and executed on three processors. Figure 2 shows a task *b*, replicated on three processors, with its output being used by a task *a*, of which only one replication is shown. The output of each replication of task *b*, a tuple of one or more words, is placed during execution in an output buffer in that processor. Subsequently these results are copied from the output buffer into the output region of the data file and are broadcast to all the other processors. At each of the processors, the various replications of the results of task *b* are received in the regions of the data file corresponding to input from the various processors executing task *b*. In each of the processors, the three or five versions of the results from task *b* are extracted from the data file by voting software and the majority result is placed in the input buffer, from whence it can be obtained by any task that needs to use the results of task *b*. If there is no majority, a distinguished value is placed in that buffer and any special action is then the responsibility of the tasks using that value. All results broadcast are voted in every processor, even though possibly no task on that processor will use the voted value. Since voting takes time, the various words that are components of the result of a task may be voted at different times.

The voting software notes any discrepancies amongst the values on which it votes. A task *error reporter*, run periodically on every processor, generates a synopsis of the errors detected on that processor and broadcasts the synopsis, as is shown in Figure 3. The *global executive* task, which is replicated like other critical tasks, receives the error synopses broadcast from the various processors and decides from them which processors are faulty. The global executive is responsible for the reconfiguration of the system, generating the configuration of processors to be used, excluding the processors deemed faulty and distribute the execution of application tasks appropriate to the current phase of the flight among the configured processors. In each processor, the results from the various replications of the global executive are voted and then used by the *local executive* task to select a task schedule for its scheduler, and to set up the sets of processors executing each task for use by the voting software. Note that, while the global executive task is a replicated and voted task common to the whole system, the error reporter and the local executive are tasks specific to each processor individually and their results cannot be voted. Even though they are run on every processor, the results they generate relate to their own processor alone. Care is taken in the design to ensure that errors in the results of an error reporter or local executive can damage only its own processor.

The schedule for SHFT is designed so that different combinations of tasks can be executed on different processors. Replicated tasks can be executed at different times on different processors therefore. Figure 4 shows a small part of an activity sequence on three processors. The schedule is organized into equal *subframes*, which would typically be one or two milliseconds long, and are triggered by interrupts from each processor's clock system, the only interrupts in the SHFT system. The sequence of activities to be performed by a processor within a subframe is determined by a schedule table, which is selected from several such tables by the configuration broadcast by the global executive. Within a subframe, the schedule can require a sequence of broadcasts, votes, and task executions. Voting and task execution require the main processor and thus the time required for them is constrained by the length of the subframe. The broadcasting mechanism of the broadcast interface operates autonomously from the processor. A task must have completed its execution before its results can be broadcast, often in the next subframe, and voting of those results can begin in the subframe after that in which the last of the three or five replications are broadcast. A task execution can use results voted earlier in the same subframe, but the voting of results broadcast earlier in the same subframe is prohibited. This design decision was made to avoid a complex asynchronous proof that the result will have been received before it is voted. The overhead associated with the handling of the clock interrupt, together with the control exercised over the skew between clocks, is sufficient to ensure that results broadcast by a processor in one frame can safely be voted at any time during the next frame by any other processor.

Many of the flight control tasks require the same iteration rate, typically 10 to 20 iterations per second, but other tasks can be run less frequently. This interval, within which these important flight control functions run, is known as the *system frame*. Other tasks such as the global and local executives are also run within the same frame. Slower tasks, such as navigation tasks, are constrained to run in frames that are simple integral multiples of each other and of the system frame. An *execution window* for each task is the interval of time within which the task must be executed and its results voted. The stability of the control laws mechanized by the flight control programs depends on avoiding long transport delays between the reading of sensor values and the commanding of actuator positions. For faster flight control tasks, the execution window may be only a few subframes; slower tasks are less demanding and the execution window for them is the whole of their longer frame.

The validity of the majority voting approach depends on all task replications on working processors generating identical results, which in turn depends on these replications performing identical calculations on identical inputs. Where a task obtains inputs from other tasks run at different iteration rates, the design must ensure that all replications of the task obtain their inputs from the same iterations of the other tasks. In SHFT, this is ensured by a system involving auxiliary input buffers to preserve input values for use by slower tasks, and odd/even double buffering to ensure that a task's inputs remain unchanged throughout a frame even though the next set of input values are generated and voted at some time during that frame. Provided that the system remains safe, majority voting of the results of replicated tasks suffices to ensure that all working processors obtain the same values for the results of those tasks. Where an input is obtained from an unreplicated source, no such assurance applies. Not only may the result obtained from an unreplicated source be erroneous, which the tasks using that value may be able to accommodate, but the faulty source might broadcast different values to different processors, thus causing replicated tasks on those processors to obtain different results, destroying the utility of the majority voting. In SHFT, a mechanism called *interactive consistency* [3] is used to ensure that all working processors obtain the same value for any input derived from an unreplicated source, whether that be an unreplicated application task, a sensor, or an error reporting task.

4. An Outline of the Model Hierarchy

Figure 5 shows an outline of the various models and analyses that are used in the justification of the reliability of SIFT. Before the individual models are described in detail, we give a description of their intent and interaction. On the right of the figure is a hierarchy of models of the correct functional behavior of SIFT, while on the left are a set of analyses that yield the probability of that correct behavior. The models at the bottom of the figure describe the hardware of SIFT, upon which the more abstract analysis is based.

The models on the right of Figure 5 describe the intended functional behavior of the SIFT system. They form a hierarchy of models, with the actual binary representation of the running SIFT programs, the *BDX930 Program* as the base of the hierarchy. Each of the models above that is an abstraction of that model, omitting some of the detail present in those actual programs and thus easier to describe and understand. The highest model of the system, the *I/O Model*, describes the functional behavior that we desire from the system when it is working correctly, and is thus the model of the system that we must demonstrate is indeed consistent with the *BDX930 Program*. The intermediate models of the hierarchy are necessary so that the relationships between models are simpler and easier to substantiate formally.

The *I/O Model* specifies SIFT as a system that mechanizes a transfer function (defined by the application programs) between the sensor inputs and the actuator outputs of the system, provided that an uninterpreted predicate *system safe*, and its components *task safe*, remain true. The model defines the input/output function performed by each application task, specifying that inputs be read and the outputs generated at the right times. The model contains no description of processors, replication of tasks, or voting. The mechanisms for obtaining reliability have been completely abstracted out of the description, and all that remains is the intent -- a reliable system to fly the aircraft.

The next model, the *Replication Model*, augments the *I/O Model* with the concepts of processors, replication, and voting, and also the knowledge of which processors are working. This allows us to derive the predicates *system safe* and *task safe* from the poll sets of processors whose results are to be voted for each task. However this model contains no concept of resource allocation or scheduling.

The *Broadcast Model* increases the detail to include the concepts of resource usage and the allocation of resources through a schedule. It must therefore include a much finer grain representation of time, and indeed describe the slight time skews between processors and the clock synchronization constraints that must be met by the implementation. This is the only model that describes asynchrony between processors; the more abstract models use the same time for all processors, while the more detailed models describe single processors in isolation.

The *Denotational Model* is the first complete model of the system, described as a set of recursive functions. It could in principle be executed by an appropriate machine. Its purpose is to provide a complete specification of the behavior of the various programs in the SIFT system against which the validity of the actual implementation can be demonstrated. The various programs that form the SIFT suite are written in Pascal and form the *Pascal Implementation*, from which is derived by compilation the *BDX930 Implementation*. This is the lowest level specification of the SIFT software. In section 10 we discuss the hardware specification levels.

The functional behavior described by the *I/O Model* is assured only so long as the predicate *system safe* remains true. The analyses shown on the left of Figure 5 provide the probability that *system safe* will remain true and hence that the desired functional behavior will continue. The *BDX930 Fault Model* describes the rates of occurrence of various kinds of fault behavior, distinguishing only between faults that cause the same erroneous results to be seen and reported by all other processors, and those that cause different results to be seen and thus cause conflicting error reports that could confuse the global executive.

The *Error Rate Analysis* is used to determine the rates at which faults will cause errors, the rates at which those errors will be detected, the probability that the error reports are clear enough for the Global Executive can be certain of its diagnosis, and the rates at which the system can be reconfigured in order that the last vestiges of erroneous results can be removed from the system by the majority voting.

Finally the *Reliability Analysis* computes the probability that *system safe* remains true for the 10 hour flight duration, as processors become faulty and are reconfigured out of the system. Both the *Error Rate Analysis* and the *Reliability Analysis* are Markov models, whose state space must be demonstrated to be an abstraction of the states of the *Replication Model*, but whose transition rates are determined by the simpler probabilistic models.

5. Input/Output Model

The Input/Output model of SIFT, the highest level model specifying functional behavior, defines the input/output characteristics of tasks performed by SIFT. The model, specified axiomatically, defines the configuration of system tasks and expresses the flow of information between tasks. Based on an abstract notion of time, which may be interpreted as subframe time, we refer to iterations of a task taking place during various time intervals. The time interval for a particular iteration of a task is referred to as its *execution window*, having a beginning time and an ending time. Each task uses as inputs the values produced by its input tasks and produces one or more outputs during its execution window. Based on a high-level predicate specifying whether a task is *safe* during a particular iteration of a task, the model defines that a task which is safe during an iteration will produce exactly one output value, computed as a function of its input values. Provided that the entire system is safe throughout some interval (i.e., that all tasks are safe for that interval), we can prove by induction that all tasks will compute correct functions of their intended inputs. This defines at a high level what it means for SIFT to function correctly.

Conspicuously absent from this model is any notion that a task is replicated and computed on a set of processors. At a lower level, we shall explain that the value the I/O model defines as resulting from a given task iteration will actually be the outcome of a majority vote of processors assigned to compute the task. The *task safety* predicate taken as primitive in the I/O model, defining when a task can be relied upon to produce correct results, will be defined at a lower level to be a function of the amount of task replications and the number of working processors.

Briefly, the model is organized as follows. Each task a in the set of all executive and application tasks $Tasks$ computes a (mathematical) function A of its input values. $Inputs(a)$ denotes the set of tasks providing inputs to a . Recall that tasks do not all have the same iteration rates. For task $b \in Inputs(a)$, the most recently completed iteration of b prior to the execution window of the iteration of a , provides the input to an iteration of a . A derived function b to i of a denotes the iteration of b providing input to the i -th iteration of a . During each iteration i of a task a , $a(i)$ denotes the set of output values which may be produced. In order to map task iterations to subframe time, the function i of a is used to denote the time interval $[t_1, t_2]$ comprising the execution window of the i -th iteration of a . The functions $beg(i$ of $a)$ and $end(i$ of $a)$ are used to denote the beginning and end of the execution window, respectively.

The overall structure of task configurations within the I/O model is illustrated in Figure 6 shown below. For a task such that the predicate *a* safe during *i* is true, *a* will produce exactly one output value during its execution window. A task which is not safe during its iteration may produce any number of outputs. Because the configuration of tasks is different for different phases of the flight, not all tasks necessarily compute each iteration. An uninterpreted predicate *a* on during *i* determines whether *a*(*i*) is expected to compute a function of its inputs or to return a special \perp element as its value.

Within the I/O model the interactive consistency algorithm is defined as a special form of task. For such a task *a*, satisfying the predicate *i/c*(*a*), its associated function *A* is the identity function. Recall from our discussion in Section 3 that the interactive consistency algorithm is used in order for multiple processors reading unreplicated (and possibly unstable) input to reach agreement on an input value. As we explain below, a safe interactive consistency task will always produce a single output value.

Based on these primitive functions and predicates, the I/O model contains seven axioms, expressing constraints on the schedule defining when task iterations are to take place and that safe tasks compute functions of their designated inputs. We do not illustrate the entire set of axioms here. The axioms related to the scheduling of task iterations are straightforward. They express basic requirements that successive iterations of a task are properly ordered in time and that the execution window of a task *b* must precede the execution window of a task *a* to which it provides input.

The major axiom defining the Input/Output behavior of a task is the following:

$$\begin{aligned} &\forall a \in \text{Tasks} \quad \forall i \quad \forall v \\ &\quad a \text{ on during } i \wedge \text{task } a \text{ safe during } i \wedge \\ &\quad \forall b \in \text{Inputs}(a) \quad b(b \text{ to } i \text{ of } a) = \{v_b\} \\ &\quad \supset \\ &\quad a(i) = (A(v_{\text{Inputs}(a)})) \end{aligned}$$

where $v_{\text{Inputs}(a)} \equiv \{v_b \mid b \in \text{Inputs}(a)\}$. This axiom defines that any iteration of a task *a*, such that (1) *a* is both on and safe and (2) each task *b* providing input to the *i*-th iteration of *a* returned exactly one output value v_b during its corresponding iteration, will return exactly one output during its iteration. The value produced will be that resulting from applying its designated function *A* to the set of values produced by its input tasks. Thus, provided *a* is safe and its input is stable, it will correctly compute an output value.

In the case of interactive consistency tasks, one additional axiom governs its input/output characteristics:

$$\forall a \in \text{Tasks} \quad \forall i \quad \exists v \quad (i/c(a) \wedge \text{task } a \text{ safe during } i) \supset a(i) = \{v\}$$

This defines that an interactive consistency task which is safe during its iteration will always produce a single value as output. By the previous axiom, if its input task is safe and thus provides a single output, the interactive consistency task will perform its associated function (in this case the identity function) on the input. Even if the input task is not safe however, the current axiom defines that *some* output value will be produced.

These are the major axioms of the I/O model. In the next section, we present the next lower-level model and show how the primitives and stated axioms of the I/O model are supported at the next level.

6. The Replication Model

The axiomatically-specified Replication model, at the next lower level, introduces the notion that tasks are replicated and executed by some number of processors. Based on a high level concept of each processor communicating its results to all other processors, a specification of the majority voting performed by each processor is given. Also defined is the information flow through which error reports from individual processors are provided to the global executive. This information is used by the global executive in order to diagnose processor faults and remove from the configuration processors deemed to have solid faults.

The concept of task scheduling has been refined to define not only the execution window for task execution but also the set of processors assigned to execute the task. The function *poll* for *i* of *a* denotes the set of processors assigned to compute the *i*-th iteration of task *a*. The I/O model primitive predicate *a* on during *i* is derived within the Replication model as $\exists p \in \text{poll}$ for *i* of *a*.

With the concept that a processor computes an iteration of a task comes the primitive function *a*(*i*) on *p* which denotes the set of outputs produced by processor *p* for the *i*-th iteration of task *a*. In a manner left unspecified by this level model, processor *p* communicates its results to all other system processors. The primitive function *a*(*i*) on *p* in *q* denotes the set of values that processor *p* has reported to processor *q* for the *i*-th iteration of *a*. A derived function *a*(*i*) in *q* is used to define the result of processor *q* voting on the output of the *i*-th iteration of *a* based on the results communicated to it. As we shall show shortly, the I/O primitive *a*(*i*) for a safe task iteration will be derived as the value a majority of assigned processors obtained by their voting. All processors are required to report the results of each task computation to all processors, and all processors are required to vote on all received values.

One other newly introduced derived function appears in the Replication model. The DWindow for *b* to *i* of *a* is defined to be the data window, consisting of the time interval starting at *beg*(*b* to *i* of *a*) of *b* and ending at *end*(*i* of *a*). Based on this function, we define DWindow for *i* of *a* to be the interval extending from the beginning of the execution window of the earliest input task to *a* and extending to the end of the execution of *i* of *a*.

The overall structure of the Replication model is illustrated Figure 7. The task structure shown is a refinement of the task configuration illustrated in Figure 6.

With the concept of processor computation occurring in the Replication model, the task safe predicate appearing as primitive within the I/O model can now be derived within the Replication model in terms of working processors. The Replication model includes an uninterpreted variable *S* which denotes the set of properly functioning processors at any given time. $S^{[t_1, t_2]}$ denotes the set of processors properly functioning during the interval $[t_1, t_2]$. This variable must remain uninterpreted in all lower level models as well, since the implementation will never have perfect information concerning the set of correctly functioning processors. Using this concept

of the set of working processors, we can now derive the task safe predicate of the I/O model as follows.

task a safe during $i \equiv$

if $\text{not}(a)$:

$(2 \times |\text{poll for } i \text{ of } a \cap S^{\text{Window for } i \text{ of } a}|) > |\text{poll for } i \text{ of } a|$

$\vee \neg a \text{ on during } i$

if $\text{vc}(a)$: ...

In the above definition, $|s|$ denotes the cardinality (i.e., number of elements) of the set s . The definition states that a task is safe either if a majority of the processors assigned to compute the task are working for the data window of the task or if the task is not on during i . It is necessary that the processors correctly function for the entire data window of the task in order that we can be assured that the processor will not corrupt its input data prior to its use. We omit discussion of the conditions necessary to define the safety of interactive consistency tasks.

Based on these concepts, we can now define derived functions $a(i)$ in q and $a(i)$. Definition of the latter function will provide the mapping up to its use as a primitive function in the I/O model. $a(i)$ in q is defined by:

$a(i)$ in $q \equiv$

if $q \in S^{\text{Window for } i \text{ of } a} \wedge \text{task } a \text{ safe during } i$

then $\text{maj}(\text{bag}(a(i) \text{ on } p \text{ in } q : p \in \text{poll for } i \text{ of } a))$

In the definition, bag is a function creating a bag¹ with the specified elements, and maj is a function returning a (singleton) set containing the majority value of its singleton set arguments. This definition defines that the value a working processor q obtains for a safe task a will be the value reported to q by a majority of the processors assigned to compute the i -th iteration of a . From the definition of task safe, one can see that task safety implies a majority of working processors assigned to compute the task. Because our voting axiom (given shortly) will ensure that a working processor will produce only a single output during its execution window, we can be assured that the majority of the singleton sets reported for a safe task will indeed be the majority computed by all assigned processors.

We now define the derivation of $a(i)$ as:

$a(i) \equiv$

if task a safe during i

then $a(i) \text{ on } p : p \in (\text{poll for } i \text{ of } a \cap S^{\text{Window for } i \text{ of } a})$

The value of the i -th iteration of a safe task a used in the I/O model is thus the singleton set that any working processor assigned to compute i of a obtains through voting. We are guaranteed (and can prove as a theorem) that all such processors will obtain the same result.

With these functions and predicates, the axioms of the Replication model can be stated. The model consists of ten axioms. By using the derived functions and predicates, many of the axioms appear identical to those of the I/O model. The difference is of course that each axiom is expressed in terms of Replication model primitives rather than I/O model primitives. Our main execution axiom in the Replication model is:

$\forall a \in \text{Tasks} \forall i \forall p \in (\text{poll for } i \text{ of } a \cap S^{\text{Window for } i \text{ of } a}) \forall v$

$a \text{ on during } i \wedge \text{task } a \text{ safe during } i \wedge$

$\forall b \in \text{Inputs}(a) b(b \text{ to } i \text{ of } a, \text{ in } p = \{v\})$

$\supset a(i) \text{ on } p = A(v_{\text{Inputs}(a)})$

where $v_{\text{Inputs}(a)}$ is defined as in the Replication model. This axiom, quite similar to its counterpart in the I/O model, defines that a working processor p will compute the proper function of its input values for a task a , provided a is safe. The interactive consistency axiom is exactly as given in the I/O model.

Also included in the Replication model are axioms defining the error reports that each processor must file when discrepancies are discovered during voting. Each processor contains a special error reporting task err . Any working processor p which detects that the value a processor q reported for the i -th iteration of a task a is required to submit an error report via the processor's error reporting task. Only under these circumstances can a working processor report discrepancies. The error reporting tasks in turn provide the reports as input to the global executive. In the Fault Diagnosis model we specify the algorithm employed by the global executive in its determination of who is at fault and whether a solid or transient fault has occurred.

Using the derivations for the primitives of the I/O model that we have given, one must show that each axiom of the I/O model is provable as a theorem within the Replication model.

¹A bag is a "set" which can contain multiple copies of the same element.

7. The Broadcast Model

The axiomatically specified Broadcast model occurs at the next lower level in the specification hierarchy. At this level, a more explicit model of the actual scheduling of *broadcast*, *voting*, and *task execution* activities is introduced. While the Replication model defined the effect of the communication between processors and of task execution, it did not define the means by which this is achieved. The current model defines the sequence of activities, derived from the schedule table, which is to support the specified effect. The $\alpha(i)$ on p in q primitive function within the replication model is refined to define the broadcast mechanism responsible for communicating the value of $\alpha(i)$ on p to each of the other processors. Based on the specification of the activity schedule present for each processor, specific requirements on the scheduling of the information flow through the system are formulated. Among these requirements are (1) each iteration of each task is scheduled sufficient execution time, (2) the broadcast of each task output is performed within the required period and after the completion of the task iteration, (3) voting on each task result occurs within the required frame, after the broadcast has been received and prior to its use as input, and (4) all activities scheduled for a given subframe have sufficient time to complete.

As we explained in Section 3, four different kinds of activities are involved in the operation of SIFT. Within the Broadcast model these are represented as

- <Overhead>
- <"Broadcast", a >
- <"Vote", a, s >
- <"Execute", $a, start, finish$ >

The "Overhead" activity represents the overhead period occurring at the beginning of each subframe, and the "Broadcast" activity initiates the asynchronous broadcast of the output of task a to all processors. Recall that voting on a task result may occur in stages. Within this model we explicitly represent the output of a task as a sequence of values (based on the number of machine words representing the result). A particular "Vote" activity votes on a subsequence s of the output sequence resulting from the execution of task a . The specification of the "Execute" activity for a task a includes an indication of whether this is the *start* of the task iteration, an intermediate execution, or the *finish* of the iteration.

The primitive function $sched(c, t, p)$ within the model denotes the schedule table, specifying the sequence of activities to be executed by processor p during subframe time t when in configuration c . As discussed in Section 3, the configuration at a particular time consists of a mapping from each task to the set of processors which have been assigned to execute the task. The configuration is calculated once per frame by the global executive and broadcast to all processors. The function $config(t, a)$ denotes the set of processors in the configuration for task a at subframe time t . The *poll* for i of k primitive function in the Replication model is mapped up from the Broadcast model as $config(beg(i$ of $a, a))$, i.e., as the configuration present at the beginning of the execution window for the i -th iteration of task a .

From the schedule table $sched$ the actual schedule of activities for each processor is determined. During a subframe, each processor performs the sequence of activities indicated by the schedule table. Within the model, an *activity time* t_p denotes a finer grain of time than subframe time. It is used to order the cumulative activities performed by a processor p . Two auxiliary functions are used to convert between subframe time and activity time. The function $subframe(t_p)$ maps an activity time t_p on processor p to the subframe in which the activity is performed. In the other direction, the function $start(t_p)$ maps a subframe time t to the activity time of the first activity of processor p in subframe t . Based on activity time, the derived function $schedule(t_p, p)$ denotes the activity performed by processor p at activity time t_p . This history of processor activity is derived from the schedule table and the changes of configuration mandated by the global executive.

Figure 8 below illustrates the flow of information through the system as a result of the Broadcast, Vote, and Execute activities. The functions

- output for a on $p(t_p)$
- datafile out for a on $p(t_p)$
- datafile in q for a on $p(t_q)$
- input n in q for e of $a(t_q)$

denotes the values of each of these data structures at each activity time.

The output buffer for task a on processor p is modified during the Execute activities performed for the task. As a result of a Broadcast activity on processor p for task a , the value of output for a on p is transferred to datafile out for a on p and an asynchronous broadcast is initiated. Sometime later (as we discussed in Section 3), the value broadcast is received in datafile in q for a on p within each processor q . As the result of a Vote activity <"Vote", a, s > on processor q , a particular subsequence s is extracted from the result for task a received in each of the datafile in buffers and voting, based on the designated poll set is performed. The result of the vote is placed in a set of input buffers input n in q for e of a at f , for each element e contained in the sequence s . As we explained in Section 3, because of disparate iteration rates of the tasks it is necessary to double buffer the results of the voting and to maintain separate copies of the results for each task iteration frequency depending on the value. The n parameter is the boolean value selecting the buffer for receipt of the value, while f is the frequency parameter quantified over all iteration rates for which there is a task depending on the output of task a . The choice of buffer in the double buffering scheme is accomplished using two primitive functions $wrtselect(t_p, a)$ and $rdselect(t_p, a)$ to select the appropriate buffer for writing and reading, respectively, the result for task a at activity time t_p .

Within the Broadcast model is the first indication that the SIFT system is not synchronous. Associated with each processor p is a function $real(t_p)$ which maps activity time on processor p to real time. As we discussed in Section 3, because of clock skew and transport delay within SIFT, the processors will not be synchronized. In order for the system to function correctly, it is necessary that the clocks remain within a specified tolerance of each other -- to do so is the responsibility of the clock synchronization task which is part of each processor's Local Executive. As we discussed earlier, SIFT is carefully designed so that the distributed system is *effectively synchronous*. Assuming the correctness of the clock synchronization algorithm, asynchronism caused by processor clock skew has no external effect. In the case of an asynchronous Broadcast activity, for example, our specifications define the value at the destination only after the latest time at which the broadcast could have been completed given the maximum processor skew. It is incumbent upon

us to prove that no access will be attempted of the data before this time in order to map this asynchronous system up to the higher-level synchronous Replication and I/O models.

Based on the Broadcast-level primitive functions, the high-level operations representing information flow given in the Replication model can be refined to define information flow directly relative to actual data structures present in the SIFT system. The $\alpha(i)$ on p primitive in the Replication model can be derived in terms of the value of output for a on p at the finish of the Execute activity for a on processor p . The primitive function $\alpha(i)$ on p in q can be derived in terms of the value of datafile in q for a on p at the times of the Vote activities for a 's result. The function $\alpha(i)$ in q within the Replication model can similarly be derived as the values of the input α in q for e of a at f for the appropriate buffer number n and for each element e of the result.

The execution and data windows for each iteration of each task are present within the Broadcast model and form envelopes during which certain activities must be completed. In particular, all execution activities and all processors' voting on the results must be scheduled within the execution window, and all execution windows of all input tasks must be contained in the data window for the task.

In terms of these primitives, we can now illustrate the axioms defining the Vote and Execute activities, corresponding to the definition of voting and the main execution axiom of the Replication model given in previous section. We first give the axiom defining the Vote activity.

$$\begin{aligned} & \forall p \forall a \forall s \forall i_p \forall e \in s \forall f > \text{rate}(a) \\ & p \in S^{[i_p, i_p]} \wedge \text{schedule}(i_p, p) = \langle \text{"Vote"}, a, s \rangle \\ & \supset \\ & \text{input wrtselect}(i_p, a) \text{ in } p \text{ for } e \text{ of } a \text{ at } f(i_p+1) = \\ & \quad \text{maj}(\text{bag}(v_q(e) : q \in \text{config}(\text{start}_p(i_p), a) \wedge v_q = \text{datafile in } p \text{ for } a \text{ on } q(i_p))) \\ & \wedge \text{"nothing else changed"} \end{aligned}$$

In this axiom, $\text{rate}(a)$ is a function derived from the sched table and denotes the iteration rate of task a . "Nothing else changed" is used informally here to indicate that no other data structures are affected by the Vote activity. Briefly, the axiom states that if a working processor p has been scheduled to vote on a subsequence s of task a 's output at activity time i_p , then the result, at activity time i_p+1 will be that the input buffer selected by wrtselect for each element e of s will have received the majority value from the datafile in buffers corresponding to each processor in the configuration.

For the Execute activity we have the following axiom:

$$\begin{aligned} & \forall p \forall a \forall i_p \forall \text{start} \forall \text{finish} \\ & p \in S^{[i_p, i_p]} \wedge \text{schedule}(i_p, p) = \langle \text{"Execute"}, a, \text{start}, \text{finish} \rangle \\ & \forall b \in \text{Inputs}(a) \forall 1 \leq e \leq \text{result.size}(b) \\ & \quad \text{finish} \wedge \\ & \quad \text{input rdselect}(i_p, \text{minrate}(a, b)) \text{ in } p \text{ for } e \text{ of } b \text{ at } \text{rate}(\text{minrate}(a, b))(i_p) = v_{b,e} \\ & \supset \\ & \text{output for } a \text{ on } p(i_p+1) = A(v_{\text{Inputs}(a)}) \wedge \text{"nothing else changed"} \end{aligned}$$

where

$$v_{\text{Inputs}(a)} = \{ v_{b,e} \mid b \in \text{Inputs}(a) \wedge 1 \leq e \leq \text{result.size}(b) \}$$

Despite the forbidding appearance of the axiom the behavior specified is quite simple. A working processor p which is scheduled to perform an Execute activity for a at activity time i_p will do the following. If this is the finishing Execute activity for the iteration, and if $v_{b,e}$ is the value in the selected input buffer for the e -th element in the value that processor p has voted for the result of b , then at activity time i_p+1 the output buffer for a on p will contain the result of applying a 's characteristic function A to all input values $v_{b,e}$. In doing the selection of buffer in the double buffering scheme, the function $\text{minrate}(a, b)$ is used to select the slower of the two tasks for determination of the proper buffer.

Recall that within the higher level models the interactive consistency algorithm was specified as being performed by a special task. This is refined within the Broadcast model to actually consist of a number of "Broadcast" (and in some cases "Vote") activities.

One can see from the brief description of the predicates and functions comprising the Broadcast model of SIFT and from the axioms for voting and task execution that the structures and concepts defined are approaching those employed in the implementation. Our specifications are becoming progressively more detailed and lower-level. In viewing this level of specification and all lower levels of specification the need for more abstract system models in determining system correctness should be apparent.

8. The Denotational and Imperative Models

All the models previously presented have been axiomatically specified, stating required *properties* of primitives within the model without giving complete functionality. The *Denotational Model* is the first complete model of the system. It specifies a set of recursive functions comprising the SIFT executive which is replicated on each processor. It could in principle be executed by an appropriate machine, albeit with excruciating inefficiency. Its purpose is to provide a complete specification of the behavior of the various programs in the SIFT system against which the validity of the actual implementation can be demonstrated. Consequently it is highly constrained by the needs of the program verification system.

The various programs that form the SIFT executive are written in Sequential Pascal and form the *Pascal Implementation*, from which is derived by compilation the *BDX930 Implementation*. These represent imperative models of the system, the latter of which being the actual SIFT implementation. Many programs in the system can be proved correct in their Pascal representation, and such proofs are simpler because it is easier to understand the intent of the Pascal program. The Pascal program is written within the limits imposed by a sequential programming language. As such, real-time communication via external interrupts and message passing through the data files cannot be represented within the program. Such aspects of the SIFT executive are represented through changes to special variables and flags not assigned within the Pascal program. Because of this, a few portions of the system cannot easily be verified in Pascal, for instance the scheduler and the clock synchronization code. Any verification that is attempted on the basis of the Pascal code must be predicated upon the proof that the clock synchronization algorithm allows one to reason about certain segments of the code as sequential programs with no external interference. Any such verification of Pascal functions also depends upon a proof that the translation from Pascal to BDX930 code is correct. It is therefore anticipated that the majority of the formal proof of correspondence will be between the *Denotational Model* and the *BDX930 Implementation*. Where Pascal program proofs are available, the various lemmas and invariants developed for that proof can be mapped down into the BDX930 Machine Code proofs, greatly speeding their construction.

The formal verification of the *BDX930 Implementation* is performed by reference to the *BDX930 Specification* of the behaviour of the processor and associated hardware. Developed for this purpose is a formal interpretive model for the BDX930 machine, written in Boyer-Moore recursive function theory and derived from an ISPS semi-formal machine specification. Here too design faults might lurk, and it is necessary to demonstrate the consistency of the hardware with its specification. This will be a two step procedure. First it must be shown that the *BDX930 Microprogram* executing on the *BDX930 Microprogram Processor* is consistent with the *BDX930 Specification*, and then it must be shown that the *BDX930 Logic Design* correctly implements the *BDX930 Microprogram Processor Specification*.

We do not discuss these low level models within this paper.

9. Reliability Analysis

The purpose of the various analyses culminating in the reliability analysis is to estimate the probability that the SIFT system will enter a state in which *system safe* is not true, placing an upper bound on the probability of system failure. Two primary analyses are involved, the *Error Rate Analysis*, which involves the rate of detection, reconfiguration, and error masking, and the *Reliability Analysis*, which investigates *system safe*.

The Reliability Analysis is based on a discrete Markov model containing very many states. Fortunately it is possible to perform an analytic reduction of these states, discarding states with negligible occupancy and combining together many other states until the number of states becomes tractable. This reduced Markov model has states described by three coordinates, the number of processors remaining in the configuration, the number of processors in the configuration that have solid faults, and the number of transient faults that have occurred and whose erroneous results have not yet been completely masked by the majority voting. The reduced Markov model is evaluated by successive squaring of the transition matrix. Figure 9 shows the model in the plane where the number of transient errors is zero. Note that in some of these states *system safe* can be false, where a second or third fault has occurred before the system has completed the reconfiguration from an earlier fault, or where the system has exhausted its supply of spare processors.

The validity of the Markov analysis depends on the correspondence of the state space to the states of the actual system, the approximations introduced by the analytic reduction and the evaluation method, and the justifiability of the transition rates between states. The correspondence with the actual system is substantiated by demonstrating that the states of the Markov model are an abstraction of the states of the Replication Model. Approximations introduced by the reduction and evaluation can readily be shown to be negligible. The justification of the transition rates, which are derived from the Error Rate Analysis, is less easy. A Markov analysis requires that the transitions are independent of each other and satisfy a Poisson distribution. However, it is clear that these requirements are not completely or even substantially, satisfied for the recovery transitions. At present we depend on a subjective assessment that the actual distributions are reasonably approximated by the assumed Poisson distributions, and on sensitivity analyses that demonstrate the effects of different assumptions for those distributions. Figure 10 shows an example of the results from the Reliability Model.

The Error Rate Analysis is a similar Markov model, whose purpose is to investigate (1) the rate at which faults cause errors to be generated, (2) the rate at which such errors are detected, (3) the rate at which the Global Executive algorithm can diagnose the errors and reconfigure the system, (4) the probability that the error reports are simple enough for the Global Executive to be certain of making a correct diagnosis, and (5) the rate at which the erroneous information generated by the fault is masked by the majority voting. This last rate is, of course, the rate of importance to the Reliability Analysis, for it determines the rate at which the system becomes immune to a further fault. As above, it is necessary to demonstrate that the state space for the Error Rate Analysis is an abstraction of the states of the Replication Model, and that the behavior of the algorithm for the Global Executive corresponds to that implemented.

Much of the interest of the Error Rate Analysis concerns the algorithms used by the Global Executive to diagnose transient faults and faults that generate conflicting error reports, which therefore must be represented by the BDX930 Fault Model. Transient faults, which generate errors for only a short period of time, and which may be sufficiently frequent to be a significant factor in the reliability of the system, are easy to represent. Conflicting error reports can be generated by two or more faults or by a single fault in the broadcast interface which causes a broadcast result to be seen differently by other processors. A single, very malicious fault of this type could persuade a naive Global Executive to discard a succession of good processors until the system fails, indicating that accurate analysis of such faults is essential. The great majority of faults, not involving the broadcast interface, are not differentiated for the analysis makes no assumptions about the behavior of a failed processor, allowing it even to generate entirely correct results, provided only that it remains uncorrelated with other faults. Our proof of correct operation, while *system safe* is true, is required to be sound for any form of behavior by a faulty processor, and should therefore be valid for the behavior of actual faulty processors.

10. Current Status of Project

At the time of writing this draft of the paper, the major portion of system specification has been completed and the verification effort has been initiated. The axiomatically specified I/O and Replication models have been completed and the Broadcast model is approximately 70% complete. We are in the process of translating an earlier SPECIAL state machine specification of the system into the denotational model expressed in Boyer-Moore theory. This is done to be consistent with our goal of using the Boyer-Moore theorem prover for our verification effort. Extensive design changes to SIFT necessitate revision of the SPECIAL specifications as well. The Pascal executive is now operational and has performed well during preliminary testing. We anticipate some amount of future modification to bring the implementation into line with our specifications. A Pascal to BDX930 compiler, developed outside SRI, is being used to translate into machine code. The SIFT hardware has been built and is fully operational in our laboratory, and a graphics-

oriented flight simulator is being developed in order to simulate complete flight control.

Work on verification of the SIFT hierarchy is proceeding on several levels. As mentioned earlier, a recursive model of the BDX930 machine has been written and early work is proceeding on verifying portions of the BDX930 code. Various parts of the Pascal code, such as the voting algorithm, have been proved correct already. We anticipate that verification of the consistency of the higher level models will first be accomplished by hand and later mechanized using the Boyer-Moore theorem prover. In preparation for the mechanical verification of the higher level models, each higher level axiomatically specified model is being translated into a Boyer-Moore denotational model. The result of this effort is to specify an abstract SIFT implementation based on the primitives employed in the axiomatic specification. When accomplished, the recursive models will be adopted as the higher-level models to avoid a consistency proof between the axiomatic and denotational models.

Our verification effort is expected to stop at the level of the BDX930 machine code. Left to be verified in order to make an "absolute" claim of correctness is that the hardware functions according to specification and that the Markov analysis is sound. Also left as an open question is whether our figures for the frequency and distribution of solid and transient faults reflect actual fault rates encountered during aircraft operation. The answer to the latter question falls outside the realm of formal proof, of course, and must be decided on the basis of empirical study.

Acknowledgments

The design, specification and verification of the SIFT project has involved nearly all the members of the Computer Science Laboratory, past and present. John Wensley lead the original design effort for SIFT and conceived the basic architecture. Also involved in the design were Jack Goldberg, Karl Levitt, P.M. Melliar-Smith, Leslie Lamport, Rob Shostak, Marshall Pease, Mike Green, Bill Kautz, and Chuck Weinstock. Formalization of the SIFT models is being done by P.M. Melliar-Smith, Richard Schwartz, and Leslie Lamport. The design of the Pascal implementation is due to Chuck Weinstock, Karl Levitt, Dwight Hare, Mike Green, Bob Boyer, and J Moore are involved in the mechanical verification effort. Jack Goldberg is the current project leader. The SIFT hardware was built by Bendix under subcontract.

The work reported here was supported by the NASA-Langley Research Center. The guidance of Nick Murray, our project monitor, is gratefully acknowledged.

References

- [1] Goldberg, J.
Development and Evaluation of a Software Implemented Fault-Tolerance Computer: SIFT Hardware.
Interim Technical Report, SRI International, Nov 1979.
- [2] Goldberg, J.
SIFT: A Provable Fault Tolerant Computer for Aircraft Flight Control.
Proceedings of IFIP Congress 80, 1980.
- [3] M. Pease, R. Shostak and L. Lamport.
Reaching Agreement in the Presence of Faults.
Journal of the ACM 27(2):228-234, April, 1980.
- [4] Weinstock, C.
SIFT: System Design and Implementation.
10th International Symposium on Fault Tolerant Computing, October 1980.
- [5] J. Wensley et. al.
SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control.
Proceedings of the IEEE 66(10):1240-1254, October, 1978.

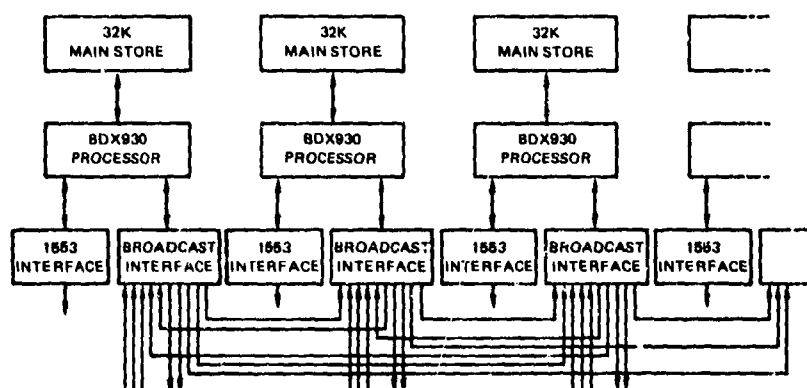


Figure 1: A View of the SIFT Hardware

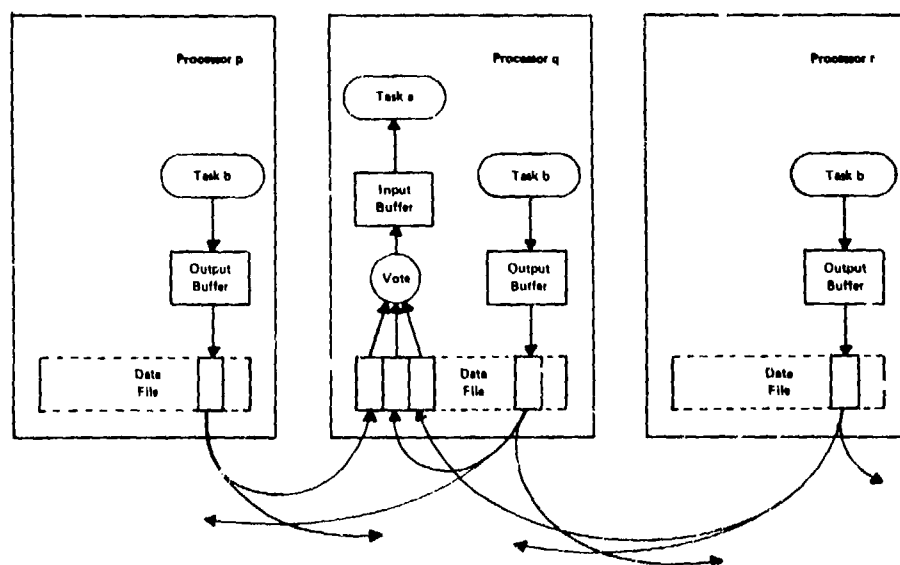


Figure 2: The Broadcasting and Voting of Information in SIFT

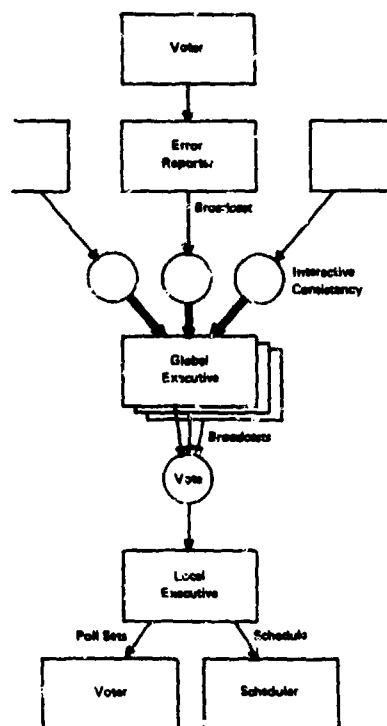


Figure 3: Information Flows for Error Reporting and Reconfiguration

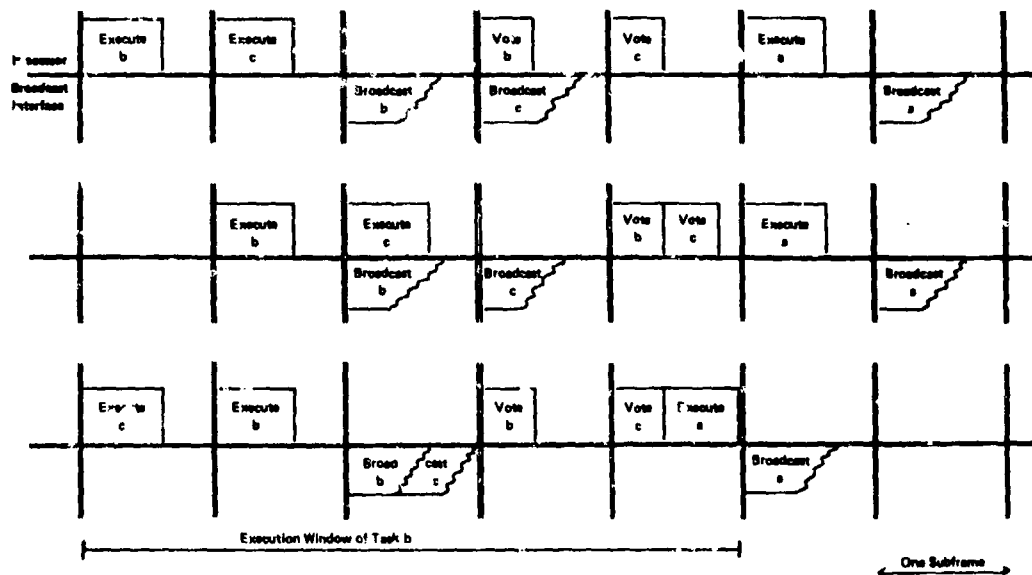


Figure 4: A Part of a Schedule for Three Processors in SIFT

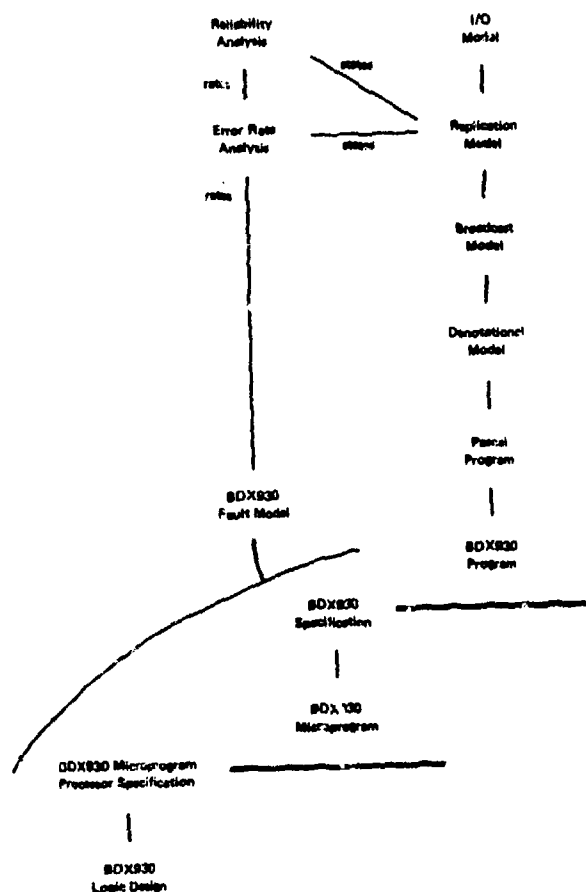


Figure 5: The Hierarchy of Models and Analyses used to substantiate the Reliability of SIFT

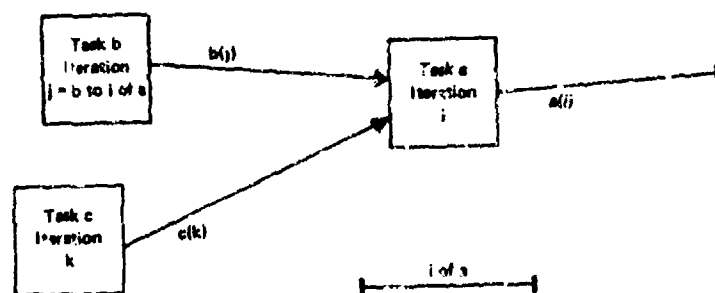


Figure 6: Three Tasks in the I/O Model

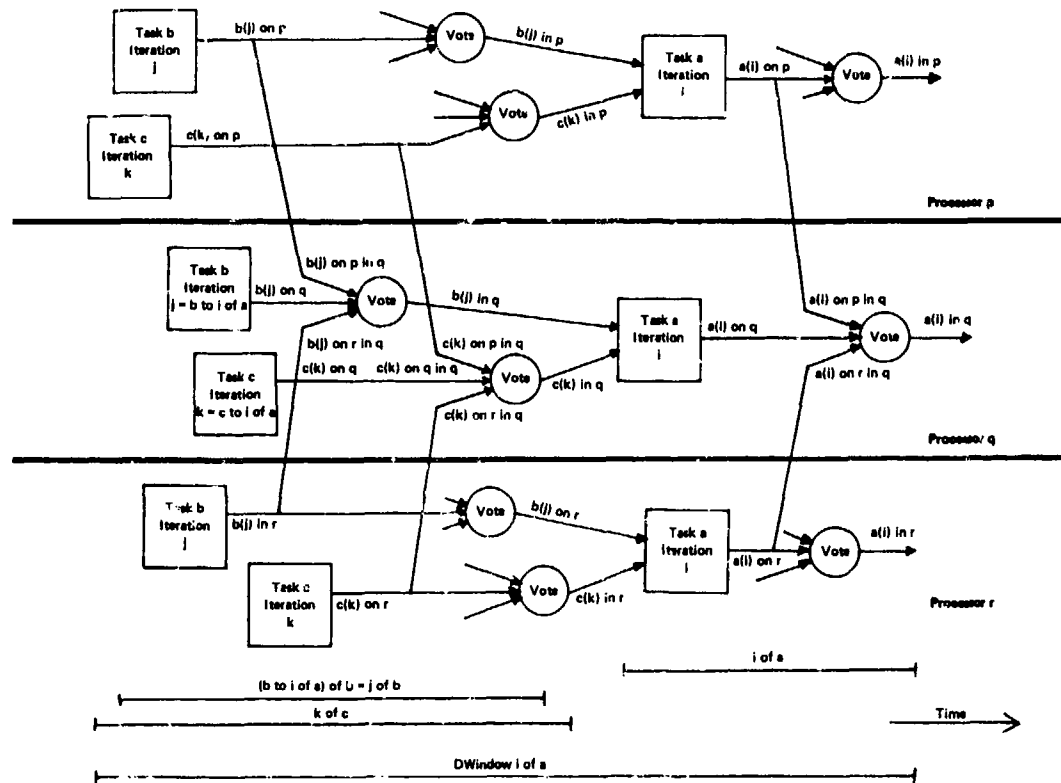


Figure 7: Three Tasks in the Replication Model

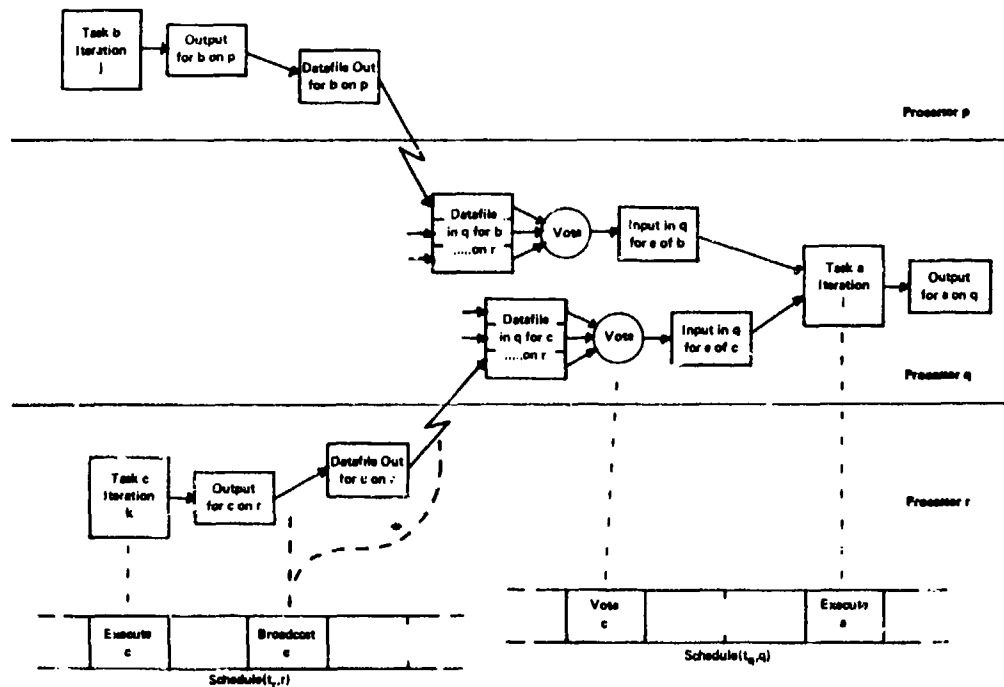


Figure 8: A Partial View of Three Tasks in the Broadcast Model

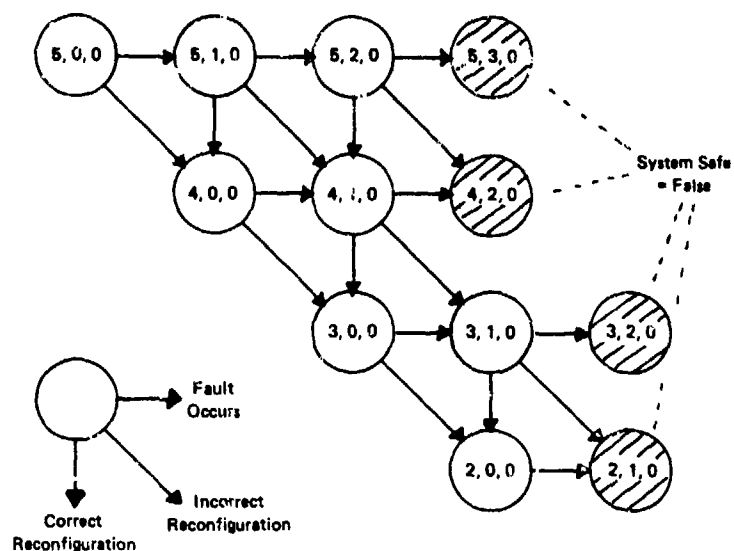


Figure 9: A Partial View of the Reliability Analysis

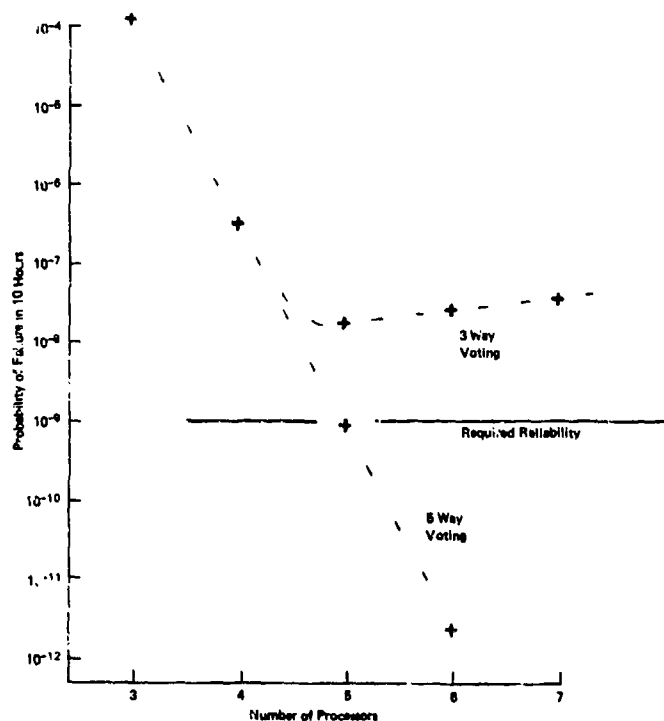


Figure 10: A Sample of the Results of the Reliability Analysis

Reconfiguration: a method to improve systems reliability

J. Szlachta

Litton Technische Werke
7800 Freiburg, W.-Germany

ABSTRACT

To improve the reliability of a flight-augmentation computer, a system with hardware and software reconfiguration capabilities was developed. The system consists of a network of n redundant computers, linked via m serial buses. A redundant computer consists of 2 CPU's, 2 memories and 2 or more I/O drivers. A fault in one of the components of the redundant computers causes a hardware reconfiguration which replaces the faulty component by its still functioning twin. If a redundant computer fails altogether, all tasks allocated to it are transferred to one of the still working computers of the network. This is made possible by loading dormant copies of the tasks into at least one other computer of the initial system. These dormant copies are periodically supplied with the program status of the active copy.

Introduction

The objective of the Redundant Computer System project was the development of an integrated hard- and software system as a basis for a highly reliable flight-augmentation computer.

The reported reliability requirements for such a computer are of the magnitude 10 failures/hour for a 10 hour flight. Values of this magnitude can only be achieved by the introduction of redundancy into the system. There are two standard methods by which this can be done. Firstly, by static

redundancy, which means by masking failures of individual components by the use of majority decisions, and secondly by dynamic redundancy, by replacing a failed component with a still functioning one of the same type.

With few exceptions like SIFT and FTMP, the majority of published systems are based on static redundancy. In the Redundant Computer System project we aimed to improve reliability by dynamic redundancy at two levels.

The first level (systems level) consists of a network of

n computers loosely coupled via m serial buses. Redundant copies of the tasks to be run are distributed over these computers.

At the second level (hardware level) the individual computers and bus links are duplicated so that a faulty component can be replaced by its twin. After a brief overview of the hardware basis at the systems level, we will concentrate our attention in this paper on the mechanics by which reconfiguration is performed.

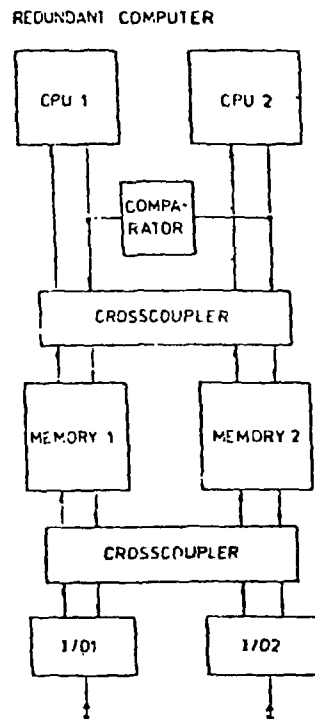


FIG 1 REDUNDANT COMPUTER

The Redundant Computer

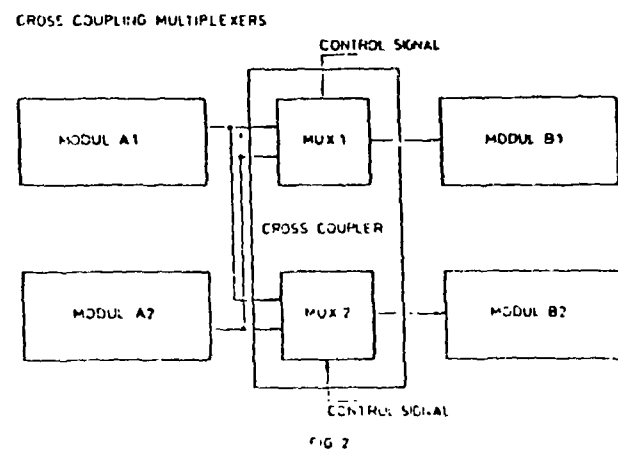
The general configuration of the individual computers of the Redundant Computer System network is illustrated in fig.1. They consist of the basic modules CPU, memory and I/O driver. These basic modules are duplicated and linked using

crosscoupling circuits. As the system is switched on, all modules are active and constantly monitored as described below. A reconfiguration to a component with a latent failure is therefore avoided.

The methods used for error detection in the different modules depends on their respective type. For the two clock synchronized CPU's the output produced is constantly compared by a comparator. If this comparator detects a difference, a self-test program is started by hardware, in order to identify the faulty CPU. The run time of this test-program is less than 1 millisecond. In a sense it replaces the third CPU of a minimum majority-system.

The error checking of the memories is performed with the use of testbits which are stored together with the data-bits. These testbits not only protect the data but also their addresses.

The error checking of the I/O drivers uses special test data and self-test loops.



The crosscoupling circuits are configured as multiple multiplexers (see fig. 2), each of which is switched by the use of individually generated control signals. A fault in one of the crosscouplers, even certain double faults, does not interrupt the dataflow between the modules.

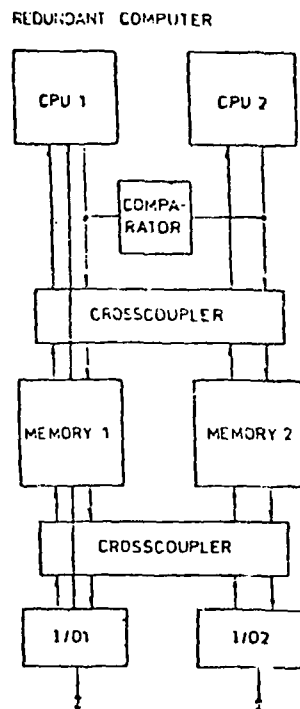


FIG 3 DATAPATH AFTER INITIALIZATION

Reconfiguration of the Computer

At the initialization of the system control signals for the crosscouplers are generated, so that CPU 1 is connected to memory 1; and memory 1 is connected to I/O driver 1. A datapath is thus created which passes over the number 1 modules (see fig. 3). The number 2 modules are producing data as well and therefore have to receive the same input as the number 1 modules, but this enters only the monitoring circuits and is blocked from the

normal dataflow by the crosscouplers.

If, for example, the monitoring circuitry detects a fault in memory 1,

it generates control signals which interrupt the dataflow between CPU 1 and memory 1, and memory 1 and I/O driver 1. Instead, a connection is created between CPU 1 and memory 2 and memory 2 and I/O driver 1. As shown in fig. 4 the datapath now passes over CPU 1, memory 2, and I/O driver 1. The reconfigured computer is still functioning, and, as far as CPU's and I/O drivers are concerned, still redundant.

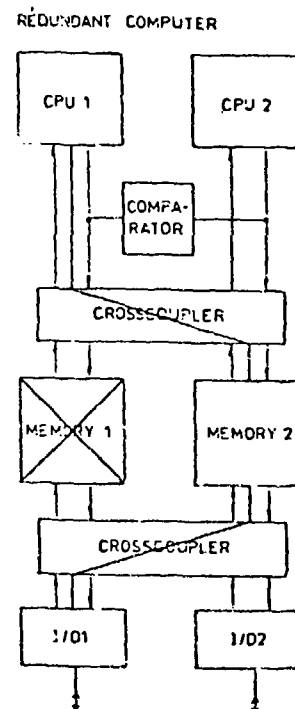


FIG 4 DATAPATH AFTER RECONFIGURATION

The systems level is informed about this event by the creation of specific status information. It can now react by the removal of critical tasks from the faulty computer

The Redundant Network

The hardware at systems level consists of a network of n redundant computers which are loosely coupled by m redundant serial buses. An example is given in fig. 5. This figure contains some additional details which refer to the demonstration model. None of the components of the network is distinguished from any of the others.

Within the limits of capacity, each of the redundant computers can perform the same function as all the others.

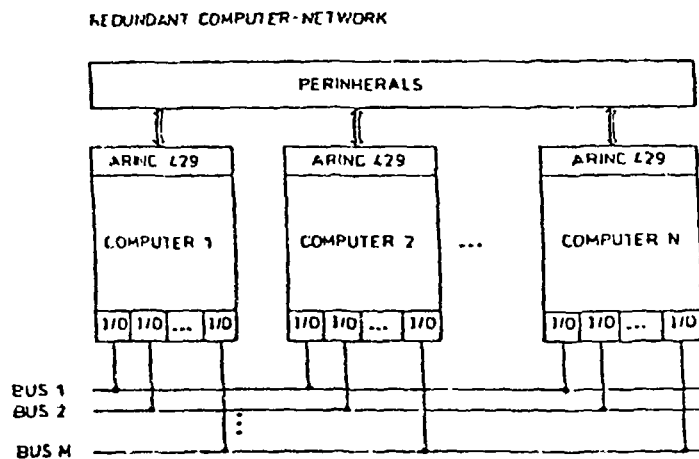


FIG. 5

Contrary to the hardware level where the monitoring of the main modules is performed by special circuitry, error detection in the computer network at systems level is done entirely by software. The main method used for error detection is activity-monitoring, that means by checking whoser each computer produces output and update messages within a predefined time interval. The purpose of update-messages will be described later.

Systems Level Software

The software at systems level consists of three types: the systemskernel, the system-processes, and the user-processes. Systemskernel and system-processes together allow the implementation of user functions independent of the system configuration and without any knowledge of the internal structure of the systems software. The actual assignment of the individual user functions to the computers of the redundant network, at any moment in time, is hidden from the user. The user processes perform the systems data processing functions. It is these user processes which the systems software attempts to keep running in case the computer to which they were originally allocated fails.

Systemskernel

The systemskernel provides all those functions which are necessary for the control of and the communication between the systems- and the user-processes. These functions are basically identical with those provided by the kernel of a normal process control operating system.

In addition however, the kernel of the Redundant Computer System contains the two functions "send update" and "receive update"

with which a process can send and receive program status information.

This program status information is used to synchronize the programstatus of redundant copies of a userprocess.

Systemprocesses

The systemprocesses perform the bulk of the redundancy-administration at systems level. They exist, like the kernel functions, as identical copies in all computers of the network. When the system is switched on, every copy of a systemprocess is active in parallel.

PROCESS CONFIGURATION

COMPUTER		1	2	3
PROCESS	1	x	x	x
	2	x	x	x
	3	x	x	x
	4	x	x	x
	5	x	x	x
	6	x	x	x
	7	x	x	x
	8	-	x	x
	9	x	-	-

FIG. 6 AFTER INITIALIZATION

The systemprocesses maintain local copies of network description data in the individual computers. This data must be kept consistent throughout the system, so that all decisions based on it are independent of the arbitrary computer on which they are performed. To achieve this, one of the systemprocesses, the configuration administrator, circulates a synchronising message through the system. Each local copy of the configuration administrator is identified by a unique

indexnumber. Using this indexnumber a neighbourhood relationship is established between the copies. The neighbour of the process with the highest indexnumber is the one with the lowest. The synchronising message is handed from one copy of the configuration administrator to its neighbour. A special algorithm is used to prevent duplication of the synchronising message and recreates it in case it is lost with a failing computer.

Besides being used for the synchronization of the network descriptions these messages are part of the error detection algorithm at systems level. The receiving copy of the configuration administrator sends an acknowledgement to its predecessor. Using this handshake protocol an activity test on the successor is performed. If this test fails the configuration administrator repeats the test with the successor of its original successor e.c.t. The results of these tests are recorded in the synchronising message. An auxiliary test-canal is used to differentiate between computer and communication failiour.

Userprocesses

The userprocesses perform the systems data processing functions proper. Copies of the code of these functions are loaded into several or all computers of the initial network. Obviously, only one of these redundant copies should, at any moment in time, take part in active dataprocessing.

Reconfiguration of User-processes

Each userprocess for which redundant copies exist must immediately after its initialization perform a call to the kernel function "receive update". In this function, the userprocess waits for the arrival of update messages. If such a message arrives, it will be accepted and its data distributed. The process returns then to the "wait for update message" state. At systems initialization, each copy of all the systemsprocesses is inside the "receive update" call, and therefore passive.

It is the function of the systemsprocess process-administrator to activate one and only one of all the copies of a userprocess. When activated, the process-administrator scans the local process configuration description until it finds a userprocess of which no active copy exists. Figure 6 illustrates such a process configuration description. For fig. 6 to fig. 9 the X indicates that a copy of that particular process is loaded into the respective computer, and a dot indicates that the copy is active.

At systems initialization this will be the first userprocess it finds. It then checks whether this process fulfills the locality criteria. If not, it continues scanning the table; if yes, it activates the local copy of the process by sending it a special type of update message. This special update message causes the receiving process to leave the "receive update" call and resume activity behind the

PROCESS CONFIGURATION

COMPUTER		1	2	3
PROCESS	SYSTEM 1	X •	X •	X •
	2	X •	X •	X •
	3	X •	X •	X •
	USER 4	X •	X	X
	5	X	X •	X
	6	X	X	X •
	7	X •	X	X
	8	-	X •	X
	9	X •	-	-

FIG 7 AFTER USERPROCESS ACTIVATION

"send update" call to which the last received update message belonged. If no update message has been received resumption takes place behind the "receive update" call. The state of the process configuration description after systems initialization is shown in fig. 7. The active copies supply all other copies with process status information by periodically calling the kernel function "send update". This call has to be parameterized with a description of all the process data which characterizes the program status at the point of call. The status of a passive userprocess copy, after receiving an update message, corresponds exactly to the status of the active copy when sending it. It is therefore possible to activate a passive copy immediately behind the send call of the last received update-message. The newly activated copy will produce

exactly the same output as the originally active one.

PROCESS CONFIGURATION

COMPUTER		1	2 FAULTY	3
PROCESS	SYSTEM 1	X •	X	X •
	2	X •	X	X •
	3	X •	X	X •
	USER 4	X	X	X
	5	X	X	X
	6	X	X	X •
	7	X •	X	X
	8	-	X	X
	9	X •	X	-

FIG 8 AFTER FAULT IN COMPUTER 2

If a computer of the network fails, another of the systems processes, the configuration-administrator, marks the column in the process configuration description belonging to the failed computer as empty. Fig. 8 shows the process configuration description after computer 2 has failed. Again, synchronising-messages are used to ensure consistency of the configuration description before the process administrator acts upon them.

The next run of the periodically-started process-administrator now again finds processes of which no active copy exists.

PROCESS CONFIGURATION

COMPUTER		1	2 FAULTY	3
PROCESS	SYSTEM 1	X •	X	X •
	2	X •	X	X •
	3	X •	X	X •
	USER 4	X •	X	X
	5	X •	X	X
	6	X	X	X •
	7	X •	X	X
	8	-	X	X •
	9	X •	X	-

FIG 9 AFTER RECONFIGURATION

Because of the loss of one or more computers, the locality criteria has changed for the now fully passive processes. It has become true in exactly one of the still running computers. As described above, this copy of the userprocess will now be activated by the local copy of the process-administrator. The result of this activation is illustrated in fig. 9. In closing it should be mentioned that this reconfiguration algorithm can also cope with the return of a computer into active service.

State of the Project

Currently we are building a demonstration model of a

redundant network consisting of one redundant and one non-redundant computer of the type LITEF 1432 linked by two twin 1553b bus connections. The systems software is nearing completion and has been tested in a simulated environment on a single computer. Integration of the system with reconfiguration tests and errorsimulations is planned for the second half of 1981.

Réseau d'Echange Reconfigurable pour Contrôle de Processus Réparti

Ch. MERAUD (SAGEM 6, avenue d'Iéna PARIS 16ème)

B. MAUREL (SAT 41, rue Cantagrel PARIS 13ème)

RESUME

Ce texte expose les résultats pour la partie procédure de l'étude d'un système d'échanges ultrafiabre à débit élevé.

Ce système doit permettre la réalisation décentralisée des échanges entre les divers équipements embarqués d'un avion ou d'un autre type de véhicule, pour l'intégration et la reconfiguration de fonctions pouvant être critiques.

L'apparition des VLSI et des fibres (2) optiques insensibles aux perturbations électromagnétiques a conduit, pour atteindre les objectifs visés, à une solution décentralisée performante par l'incorporation d'intelligence dans un module de raccordement de type universel appelé Interface Sous-Système (ISS).

Le principe retenu substitue au mécanisme traditionnellement programmé de gestion des échanges, un mécanisme dynamique immédiatement adapté aux modifications, et permettant une grande souplesse de synchronisation. Il fonctionne par diffusion de messages groupant des mots de 16 bits suivant une partition jouée en orchestre de chambre par l'ensemble des ISS répartis sur l'ensemble des équipements raccordés. Les ISS se synchronisent entre eux par extraction de l'horloge de l'émission en cours et se concertent périodiquement pour valider les échanges, commuter de mode éventuellement ou recouvrir les pannes.

La gestion des échanges au niveau de chaque équipement est donc confiée à l'ISS qu'il incorpore. Celui-ci joue sa partie spécifique en interprétant les paramètres messages que l'équipementier a inscrit dans une mémoire morte. Périodiquement, ils se donnent rendez-vous pour échanger les codes cycliques élaborés par chacun d'eux à partir des informations observées sur la ligne pendant le cycle précédent. Cette phase sert à la détection, au diagnostic des pannes, à la reconfiguration et enfin à la resynchronisation et à la réinitialisation des ISS victimes d'une panne transitoire.

Le principe substitue au mécanisme traditionnellement programmé de gestion des échanges, un mécanisme dynamique immédiatement adapté aux modifications, et permettant une grande souplesse de synchronisation.

INTRODUCTION

L'avènement au début des années 1970 des liaisons séries multiplexées normalisées (Digibus GINA, bus 1553 ...) pour l'acheminement des communications inter-équipements a permis la réalisation des premières générations de systèmes numériques intégrés de conduite d'armes ou de véhicules.

Une réflexion sur l'évolution des systèmes d'armes futurs vers tout à la fois plus de performances de complexité et de criticité ; un bilan de l'expérience acquise sur les liaisons normalisées actuelles aux possibilités limitées en regard des besoins futurs ; un examen en contrepartie de l'accroissement considérable des performances de la technologie avec l'apparition des VLSI et des fibres optiques suggèrent pour atteindre les objectifs visés une orientation nouvelle pour la réalisation de la fonction de communication vers une solution répartie par l'incorporation d'intelligence dans un module universel (ISS) de raccordement incorporé à chaque équipement.

Une recherche en ce sens a été effectuée dans le cadre d'un contrat DRET (1) par la SAGEM en collaboration avec la Société Electronique Marcel DASSAULT. Elle a abouti en décembre 1979 à une solution basée sur l'utilisation des composants de ligne des bus normalisés actuels permettant des échanges à 1 ou 2 MBd. Il faut insister sur la souplesse de cette solution sûre et immédiatement réalisable.

La solution présentée ici en collaboration avec la SAT (2) prévoit l'utilisation des fibres optiques pour atteindre les vitesses d'échanges de 10 MBd nécessaires dans les futurs systèmes.

Elle permettra alors :

- de simplifier la maintenance en réalisant toutes les communications digitales (lentes et rapides) suivant le même protocole et avec une seule famille de matériel (réseau de bus optiques redondés 10 MBd et un seul type de coupleur très intégré incorporé à chaque équipement),
- de faire survivre automatiquement la fonction de communication aux pannes et d'avoir un diagnostic précis facilitant la maintenance. La sécurité de fonctionnement pourrait dépasser 10^{-10} /heure (probabilité d'une panne non passive), et la fiabilité avant réparation pourrait atteindre compte tenu des possibilités de reconfiguration automatique 10^{-4} par 24 heures dans des conditions d'environnement sévères,
- d'incorporer des équipements en redondance pour filtrer leurs pannes et faciliter les opérations de maintenance en ligne,
- d'effectuer avec une grande souplesse pour le matériel et le logiciel, le raccordement des équipements et les modifications de configuration et d'échanges du système,
- d'obtenir une datation précise des variables échangées.

Ce système de communication, vu à travers le module de raccordement ISS, fournira aux constructeurs d'équipements embarqués un moyen d'interconnexion universel facile à interfacer et dont l'intelligence incorporée libérera les équipements des sujétions liées aux échanges.

En ce sens, il permettra l'exécution d'échanges privés et pré-définis entre boîtes d'un même sous-système afin d'en accroître l'interchangeabilité d'une application à une autre.

Dans le même but, il substituera au mécanisme traditionnel de gestion des échanges par programmation, un mécanisme dynamique non programmé immédiatement adapté aux modifications, améliorant la datation des variables et permettant une grande souplesse de synchronisation entre les équipements informatiques du système.

Pour le maître d'oeuvre et l'équipementier, il sera accompagné d'un moyen de gestion efficace pour l'intégration du système ou d'un sous-système séparé en phase de développement, sous la forme d'outils logiciels d'aide à la conception et à l'analyse permettant la vérification et l'optimisation des échanges, la documentation de chaque édition du système et la génération des paramètres de gestion des messages.

DESCRIPTION GENERALE

Composition d'une Interface Sous-Système (figure 1)

Les équipements sont interconnectés par une double liaison série multiplexée via une interface intelligent dit ISS (Interface Sous-Système).

En plus des fonctions d'interface classiques, sa fonction est :

- d'assurer la gestion des échanges systèmes ou privés, critiques ou non, sur la liaison multiplexée,
- de permettre la synchronisation des tâches de son équipement sur l'exécution des échanges,
- de réaliser la détection et le recouvrement des erreurs de transmission et leur recouvrement par reconfiguration du bus
- de réaliser la détection et le recouvrement de ses propres fautes, ainsi que sa mise hors service et son isolement du bus en cas de panne permanente.

Structure matériel du bus

Le bus est physiquement dupliqué pour survivre aux pannes. Chaque ligne est constituée de deux fibres, l'une montante, l'autre descendante.

Les dérivations sont réalisées par des coupleurs passifs transparents. En normal, chaque ISS répète l'information incidente pour que le niveau soit maintenu sur toute la ligne. En cas de panne d'un ISS, la transparence du coupleur assure la continuité de la liaison.

Structure matérielle d'un ISS (figure 2)

Un ISS est un canal spécialisé réalisé à l'aide d'un contrôleur microprogrammé dupliqué. Une telle structure permet de détecter et de passer immédiatement toute anomalie de fonctionnement pour empêcher tout comportement anarchique d'un ISS vis-à-vis du bus.

L'ISS exécute un microprogramme canal qui interprète les instructions de gestion des seuls messages intéressant l'équipement. Les instructions sont stockées dans une EPROM appartenant à l'équipement.

Il est relié aux lignes physiques du bus dupliqué par un connecteur optique et les circuits de conversion parallèle/série, modulation/démodulation, émission/réception opto-électronique qui assurent les fonctions du niveau d'interface avec la ligne.

Informations échangées

Deux types d'informations doivent être échangées : les variables périodiques et les variables aléatoires aux délais maximaux fixés. La périodicité des émissions des variables du premier type ayant leur source dans l'équipement, du point de vue des échanges elles peuvent être traitées comme les secondes pourvu que les délais d'acheminement spécifiés soient respectés. C'est ce qui est réalisé en répartissant les variables sur 4 niveaux de priorité.

Gestion des redondances

Les équipements générant des variables critiques peuvent être implantés plusieurs fois, ce qui permet une diffusion multiple de ces variables pour accroître leur disponibilité. A la réception, l'ISS de chaque équipement concerné se charge du filtrage des pannes et de l'enregistrement du seul résultat juste dans la zone de l'équipement affectée à la réception de la variable.

Ce mécanisme favorise l'interchangeabilité en permettant l'intégration redondante d'équipements standards dans les systèmes critiques sans leur imposer de modifications notables.

Tolérance aux pannes et aux erreurs de transmission

La structure interne de l'ISS assure une détection parfaite de ses propres erreurs grâce au choix d'une structure dupliquée/comparée.

Les erreurs de transmission sont détectées :

- par détection d'erreurs de modulation grâce à l'emploi d'un codage autorythmé,
- par l'emploi d'un bit de parité par mot de 16 bits échangé,
- par l'utilisation d'un code cyclique élaboré et comparé périodiquement par l'ensemble des ISS.

Lorsqu'une erreur est détectée, un basculement sur la deuxième ligne est effectué. Ce mécanisme permet de filtrer les pannes transitoires sans dégrader le système.

Un ISS détectant une erreur persistante dont il est la cause se met hors service avec une efficacité parfaite à cause de sa structure dupliquée.

Afin d'éviter l'accumulation de pannes cachées sur le bus de secours, le rôle des deux bus est périodiquement inversé afin d'être exercé par le fonctionnement normal et bénéficier d'une réparation préventive évitant la panne double.

DESCRIPTION DU FONCTIONNEMENT

Nécessité d'une grande souplesse

La gestion classique des échanges selon une trame programmée pour réaliser la périodicité des échantillonnages ou la datation des variables, est effectuée par un seul calculateur centralisé au moyen d'un programme canal écrit spécifiquement qu'il faut modifier à chaque changement de configuration du système.

Cette technique contraignante provient de la technique de programmation d'origine en automate des calculateurs. Ces derniers étaient alors uniques dans les systèmes, ce qui éliminait les problèmes de synchronisation. Utilisée aujourd'hui pour synchroniser des systèmes multi-calculateurs, elle est mal adaptée et rend toute modification laborieuse. Aujourd'hui, le fonctionnement parallèle et asynchrone des calculateurs du système exige, pour synchroniser avec souplesse les échanges de données entre les tâches multiples et hiérarchisées qu'ils exécutent, une élasticité dans l'ordonnement des échanges qu'une gestion programmée centralisée ne permet pas. Il en résulte que, pour réaliser une datation précise des retards dans des échanges rapides (par exemple inférieure au quart de période pour des variables à 20 ms), il faudra :

- soit multiplier par 4 la fréquence d'échantillonnage,
- soit extrapoler à l'émission et synchroniser les tâches émettrices et consommatrices sur la trame,
- soit transporter la donnée avec sa date de production et extrapoler sur les lieux de consommation.

Les deux premières solutions donnent une surcharge de calcul élevée et une gestion lourde ; la troisième, plus correcte, n'utilise déjà plus la trame périodique comme outil de datation.

Pour ce qui concerne la nécessité actuelle de réaliser facilement des modifications rapides de configuration du système, la suppression de la présence obligée d'un calculateur central de gestion des échanges et de ses programmes est un accroissement de souplesse notoire. Elle restitue aux fabricants de sous-systèmes, leur autonomie technique et l'initiative dans leur domaine de compétence. Les responsabilités techniques sont mieux définies favorisant la tâche du maître d'oeuvre et la fiabilité de conception de l'ensemble.

Gestion décentralisée de l'attribution du bus aux demandes d'émission des tâches

La production des variables à échanger a pour source l'ensemble des tâches réparties dans les divers équipements. Il faut ordonnancer la diffusion de ces variables pour une consommation par le même ensemble de tâches à l'initiative de chacune d'entre elles.

Pour être échangées, les variables sont groupées par train de mots de 16 bits en messages à structure fixe avec un label d'en-tête et une priorité définie à l'échelle du système. Les messages sont diffusés par les ISS de chaque équipement à tour de rôle, et identifiés en réception grâce au label.

Le problème de l'ordonnement des messages sur le bus est de même nature que celui de l'ordonnement des tâches sur l'unité centrale. Celui-ci est aujourd'hui très correctement résolu dans les systèmes temps réel multitâches modernes par un mécanisme d'activation prioritaire à partir d'événements et d'une gestion des files d'attente des tâches prêtes sur chaque niveau de priorité. La meilleure solution consiste donc à adopter ce mécanisme pour l'ordonnement des messages. On disposera alors d'une interface souple et facile entre les moniteurs temps réel de chaque équipement et la fonction d'échange. Reste à trouver une solution répartie pour l'exécution de ce mécanisme.

On y parvient en faisant exécuter simultanément le même algorithme par tous les ISS qui doivent donc fonctionner en synchronisme. A chaque instant, un ISS est en émission. Les autres sont alors en réception synchronisée par l'horloge des émissions en cours. Il y a donc une horloge commune à la population d'ISS à chaque instant qui suffit aux besoins de synchronisation.

Implémentation d'une phase de contrôle périodique des échanges

L'ensemble des messages susceptibles d'être diffusés a été réparti à l'avance sur les 4 niveaux de priorité du système. Le niveau courant reste actif tant qu'il existe dans le système des messages en attente d'émission à ce niveau, et qu'il n'est pas apparu de messages à un niveau supérieur.

Pour décider des changements de niveau et pour les autres besoins du contrôle des échanges, un dialogue entre les ISS est nécessaire. Pour limiter les retards au minimum tout en conservant un bon rendement au bus, une fenêtre de 64 mots pour un cycle de 1024 mots échangés est réservée à ces besoins.

En début de fenêtre, chaque ISS détermine parmi 4 files de 64 bits où s'affichent les messages à émettre le niveau le plus élevé de la file non vidée. Ce niveau est inséré sur 2 bits dans un mot de contrôle. Ces mots de contrôle sont ensuite diffusés successivement par les ISS selon leur ordre d'adresse physique croissante. À la fin de ces émissions et quand la fenêtre de contrôle s'achève pour démarrer un nouveau cycle, les ISS savent :

- s'ils doivent poursuivre les échanges sur le même niveau,
- s'ils doivent commuter sur un niveau supérieur et qui doit prendre la parole (elle est prise par l'ISS d'adresse physique la plus petite à ce niveau). Dans ce cas, chaque ISS sauvegarde les pointeurs du niveau interrompu pour un retour ultérieur.

En cours de cycle et en dehors de la fenêtre de contrôle, les ISS se comportent comme un canal spécialisé entre les mémoires des équipements servis et le bus. Ils n'assurent que des opérations simples de transfert de mots à l'aide de pointeurs incrémentés à chaque pas et la mise à jour du code cyclique. Néanmoins, quand l'ISS émetteur n'a plus de messages au niveau courant, la commutation vers un successeur a lieu immédiatement. Ces commutations se font vers un autre ISS soit sur le même niveau, soit vers un niveau inférieur à partir des valeurs de pointeurs antérieurement sauvegardées. Elles ne nécessitent pratiquement pas de calcul, les paramètres de commutation ayant été déterminés pendant la dernière fenêtre de contrôle.

Critère de ventilation des messages sur les niveaux de priorité (figure 4)

Les 4 niveaux sont les suivants :

- niveau 0 : des échanges différés (échanges longs, échanges de surveillance de routine, échanges de fond divers ...),
- niveau 1 : échanges temps réel ordinaire,
- niveau 2 : échanges temps réel urgents ou à fréquence rapide,
- niveau 3 : alarmes, etc.

Il est clair que la souplesse du dispositif et la facilité d'évolution dépend des marges de charges.

La figure 4 illustre le principe de répartition sur un exemple limité à 3 niveaux pour plus de clarté. En ordonnée, on classe les messages par ordre d'urgence spécifiée décroissante (courbe de droite). On calcule (courbe de gauche) la situation de pire cas de délai d'acheminement. Celle-ci s'obtient pour les de pointes de demande, en accumulant les temps de transfert des messages supposés transmis dans l'ordre de leur classement en ordonnée. La figure montre alors le principe d'une ventilation par niveau qui ménage des marges équilibrées et maximales.

Comparaison avec une trame programmée traditionnelle

La trame des échanges obtenue est très voisine de ce que l'on obtient par une programmation des échanges dans la solution traditionnelle. Les différences sont les suivantes :

- au niveau des échanges rapides la différence est insignifiante,
- aux niveaux inférieurs, l'échange est accompli dans le délai spécifié, mais avec une indétermination de positionnement croissante vers les niveaux bas due aux variations de charge des niveaux supérieurs. Ce "jitter" croissant généralement sans importance, peut si nécessaire être compensé par le mécanisme de datation fine exposé plus loin.

En contrepartie, les échanges non urgents équilibrent la charge du bus et permettent d'en exploiter efficacement toute la capacité.

La suppression de la programmation des échanges permet des modifications rapides de configuration du système conformément aux spécifications de souplesse souhaitées.

Datation fine pour le calcul des vieillissements des variables

La solution efficace à ce besoin pour les variables qui le nécessitent consiste à transporter la date d'échantillonnage dans le message plutôt que d'accroître inutilement la fréquence d'échange par rapport à la fréquence de consommation juste nécessaire. Les tâches utilisatrices peuvent alors réactualiser spécifiquement les valeurs reçues.

Pour cela, une heure système uniquement utilisable pour les calculs de retards est entretenue par chaque ISS à la cadence des échanges mots sur la ligne. Cette heure est transmise à chaque équipement par son ISS avec une périodicité spécifique (par exemple toutes les 0,5 ms, une fréquence trop élevée saturerait inutilement l'accès direct mémoire de l'équipement).

Synchronisation mutuelle des échanges et des tâches (figure 6)

Les lieux de production et de consommation des variables sont des tâches hébergées dans les équipements.

Un mécanisme souple par événements référencés, interruption ou mise en file permet de réaliser la synchronisation mutuelle tâche-message. Ce mécanisme est le suivant :

- pour l'émission, le message préparé par une tâche est signalé "prêt" à l'ISS par positionnement d'un bit d'état dans une table de 64 bits (pour un maximum de 64 messages éligibles par équipement). Il sera diffusé au plus tôt par l'ISS en tenant compte de son niveau de priorité,
- pour la réception, les messages étant systématiquement diffusés en mode label (c'est-à-dire avec un nom dans un mot de procédure en-tête), il revient aux ISS de détecter à l'aide des paramètres inscrits dans la PROM d'adaptation de l'équipement les messages qui les concernent. L'instruction de gestion fournie sur 32 bits par l'EPRROM permet à l'ISS de spécifier s'il est concerné et ce qu'il doit faire pour charger le message à sa bonne place puis insérer un numéro d'événement avec un compte rendu d'état dans une file d'attente de l'équipement. Celui-ci exploite cette file à son rythme en activant les tâches en attente sur les événements qu'elle contient.

Portée d'adressage à structure de bloc emboîtés

La portée de désignation des labels fournie par les mots de procédure est structuré en 3 niveaux :

- un niveau de commande pouvant comporter 256 valeurs (dont un petit nombre seulement est utilisé),
- un niveau de labels systèmes comportant 512 valeurs,
- un niveau de labels sous-système pouvant comporter jusqu'à 128 groupes de 256 valeurs.

Chaque ISS a la vision complète des deux premiers niveaux et des valeurs d'un seul groupe sous-système auquel il appartient. Ce niveau correspond aux échanges privés entre équipements attachés à un même sous-système. Les échanges à ce niveau peuvent être librement modifiés sans produire d'interférence entre les sous-systèmes pourvu que les marges de charges soient respectées.

La structuration avec un niveau hiérarchique supplémentaire est envisageable.

Outre sa souplesse, ce dispositif limite la capacité d'adressage nécessaire au niveau de chaque ISS à 640 valeurs permettant de limiter la capacité de l'EPRROM paramètres à 1 Kmots (de 32 bits).

SURETE DE FONCTIONNEMENT

Elle s'appuie sur une très grande sécurité de détection des anomalies de fonctionnement des ISS qui exige leur réalisation par duplication du matériel et comparaison des sorties (figure 2).

Les ISS réalisent ensemble :

- la validation des échanges en fin de cycle,
- le filtrage des erreurs dû à des transitoires,
- la reconfiguration du bus ou leur auto-reconfiguration.

Validation des échanges en fin de cycle

Au fur et à mesure des échanges, chacun des ISS élabore un code cyclique pour l'ensemble des 1024 mots du cycle.

Ce code est inséré sur 14 bits dans le mot de contrôle que chaque ISS diffuse. Le cycle est validé quand tous ces mots de contrôle sont identiques.

Les causes de pannes étant indépendantes entre les ISS, et ces derniers étant parfaitement testés par comparaison grâce à leur structure matérielle dupliquée et leur exercice continu, la validation du cycle faite par chacun d'eux, complétée par les tests de parité déjà réalisés au niveau de chaque mot échangé, est d'une sécurité pratiquement absolue.

Cette validation déverrouille l'utilisation par les tâches de l'information échangée.

Filtrage des erreurs transitoires

En cas d'erreur constatée une première fois au cours du cycle ou lors de sa validation, le cycle est affiché en faute le mode reprise est allumé et l'ensemble des ISS bascule sur l'autre ligne (ou reste sur la même en cas de perte de la deuxième).

L'ISS de plus petite valeur d'adresse qui n'a pas fait de faute, envoie en début de cycle suivant un message d'initialisation des paramètres courants qui tient en quelques mots (heure système, configuration des ISS présents, pointeurs courants).

L'ISS en faute qui avait débrayé du cycle dès la détection de l'erreur pour attendre ce message est alors resynchronisé. Les échanges non validés maintenus par les tâches émettrices sont ensuite répétés. En cas de succès, le mode reprise est effacé.

Reconfiguration des ISS ou du bus

Après trois tentatives infructueuses du même ISS, celui-ci s'éteint automatiquement. Ce signal allume un ISS de secours qui s'initialise suivant le même procédé. S'il n'y a pas d'ISS de secours, l'équipement disparaît des échanges sur le bus.

Quand plus d'un mot de contrôle se sépare des autres, l'erreur est mise au compte d'une panne de bus. En cas de faute persistante sur le même bus physique, celui-ci est abandonné.

AIDE A LA CONCEPTION

Parallèlement à l'étude et destinée à faciliter sa mise en oeuvre et son évolution, l'étude d'un outil de conception est en cours dans sa phase théorique.

Cet outil CESAR (3) (Conception Evaluation et Spécification des Applications Réparties) est un système général d'aide à la conception d'applications découpées en "boîtes noires" échangeant des messages.

Il permet de décrire :

- l'architecture logique : découpage fonctionnel en tâches échangeant des informations sous formes de messages,
- l'architecture physique : découpage en équipements ayant des caractéristiques spécifiques.

Il permet d'ajuster le niveau de détail de la description aux besoins. Une telle démarche est bien adaptée à une conception descendante de l'ensemble d'un système réparti, et permet d'offrir les services suivants :

- la vérification de la cohérence des spécifications fonctionnelles à tous les niveaux de description, soit une validation partielle et progressive durant toute la phase de conception,
- l'optimisation de l'implantation des tâches sur les équipements compte tenu des spécifications logiques et physiques,
- la documentation automatique du projet en phase de conception et en phase de développement, le niveau de détail étant ajustable,
- une modélisation mathématique du système (lorsque le niveau de détail atteint est suffisant) permettant d'étudier au moyen d'une analyse statique le respect des contraintes temporelles et la correction des spécifications de synchronisation, ainsi qu'une évaluation des performances atteintes en fonction de la charge du système.

Un tel système permettra de spécifier les échanges entre les équipements, tant du point de vue logique (nature et fonction des informations échangées) que du point de vue temporel (durée des traitements, contraintes de datation, contraintes de séquentialité, etc.). Il permettra la vérification automatique de la cohérence de ces spécifications, ainsi que la détermination de l'implantation optimale des tâches sur les équipements et celle de la répartition des messages sur les différents niveaux de priorité.

Il permettra également de vérifier le respect des spécifications de fiabilité attachées aux différentes fonctions du système, comme ten de taux de défaillance des ressources physiques mises en jeu. Un modèle markovien permettra de suivre l'évolution des probabilités des différents états du système en fonction du temps suite aux pannes.

CONCLUSION

Les principes de base du protocole ont été exposés ainsi que celui sur lequel repose la sûreté de fonctionnement et pour lequel le calcul de fiabilité prévisionnelle a été réalisé.

Aujourd'hui, l'étude à niveau du système est en attente d'une progression des résultats au niveau des bus optiques. Au-delà, une maquette expérimentale pourra être réalisée.

REFERENCES

- (1) SIERRA : Système d'Intégration et d'Echanger Réparti et Reconfigurable automatiquement.
Rapport de synthèse - Février 1980 - Contr. DRLT
- (2) Programme "Ville câblée de Biarritz" - Maîtrise d'oeuvre SAT
- (3) J.P. QUEILLE. The CESAR System : An aided design and certification system for distributed applications.
The 2nd international conference on distributed computing systems PARIS France April 08-10 80

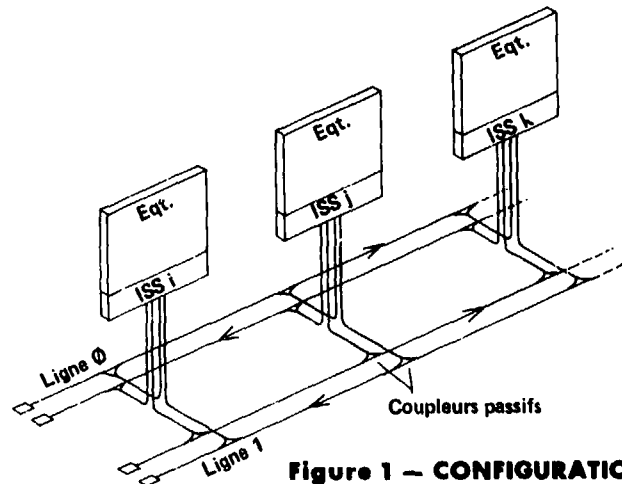


Figure 1 – CONFIGURATION D'ENSEMBLE

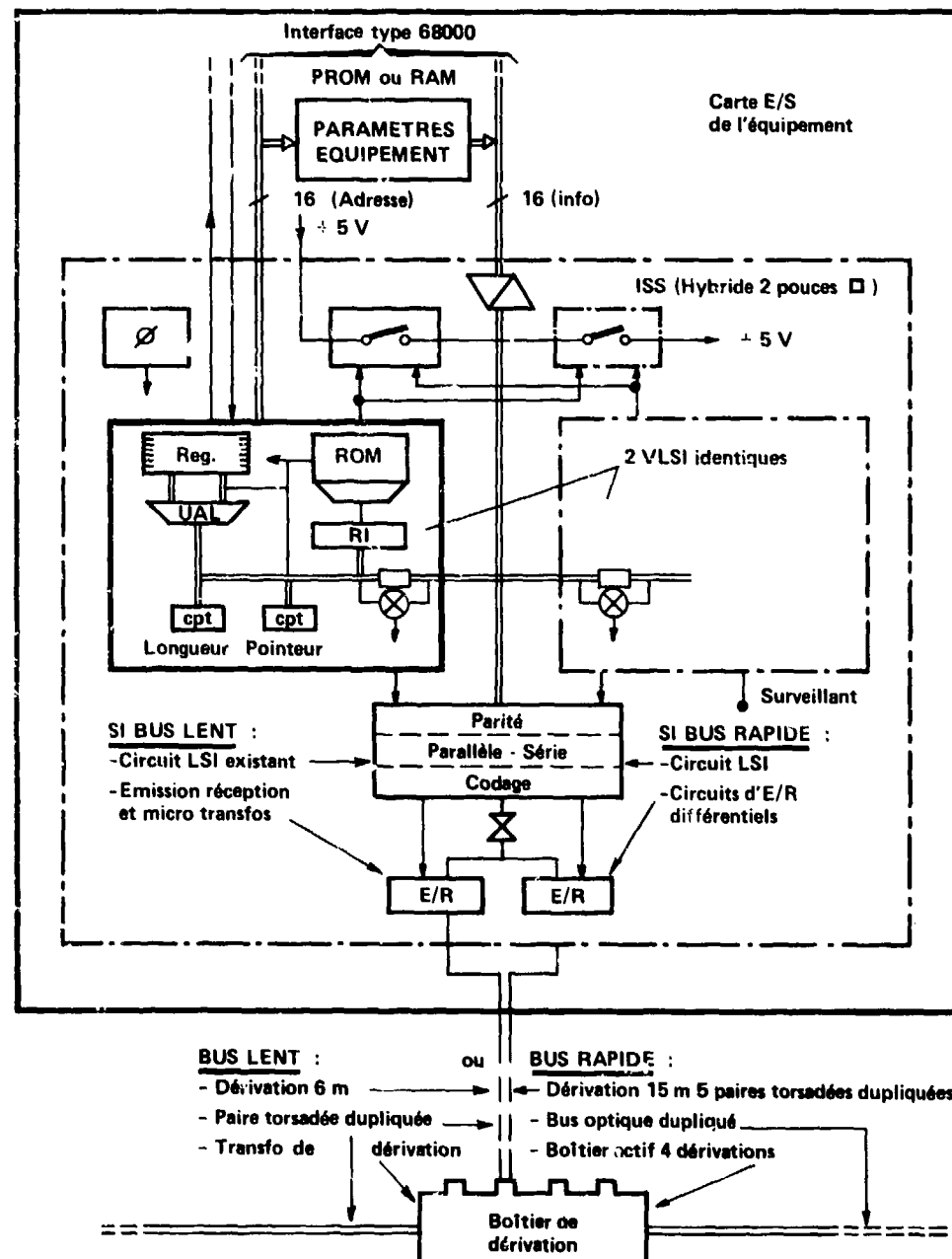
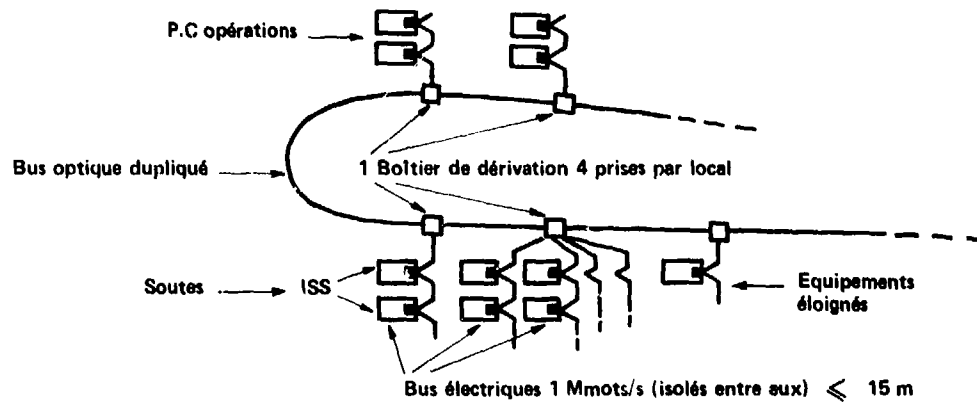


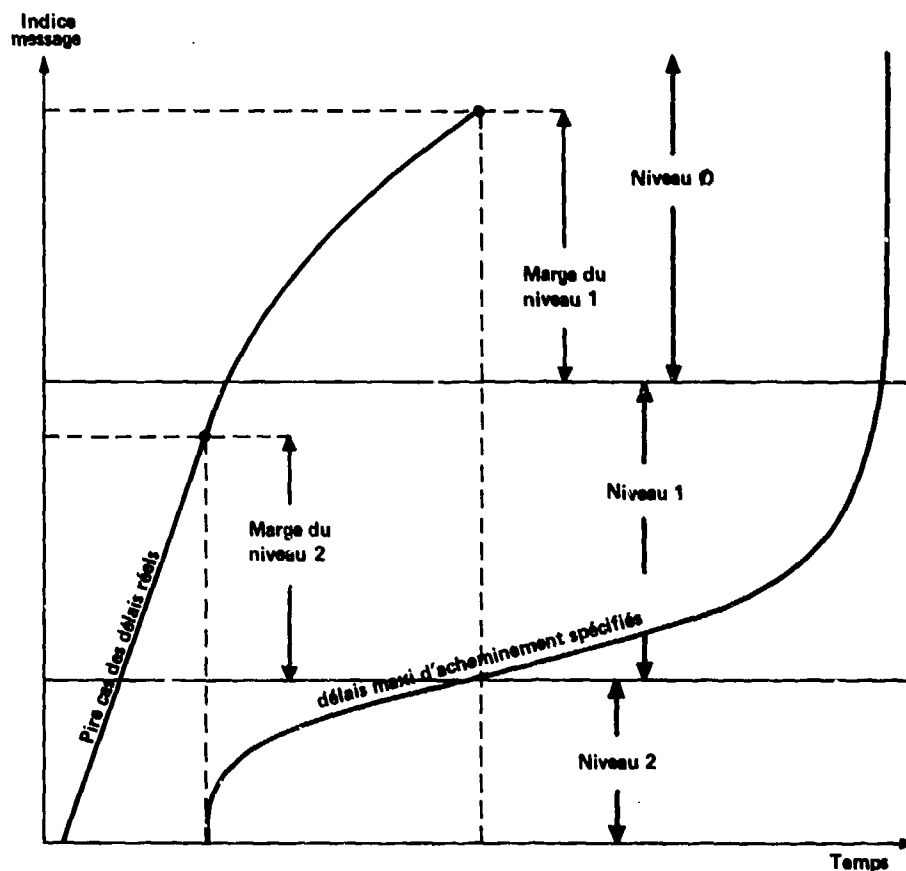
Figure 2 – ISS REPRESENTA MONTE DANS UN EQUIPEMENT



PEUVENT ETRE CONNECTES :

- CALCULATEURS
- DISQUES RAPIDES
- IMPRIMANTES RAPIDES
- ENREGISTREURS A BULLES MAGNETIQUES
- VISUS CATHODIQUES
- ETC...

**Figure 3 — SCHEMA D'INTERCONNEXION
SUR RESEAU DE DIFFUSION RAPIDE 0,5 M MOTS/S (10 M BITS/S)**



(Exemple à 3 niveaux)

Figure 4 — REPARTITION EN NIVEAUX DES MESSAGES

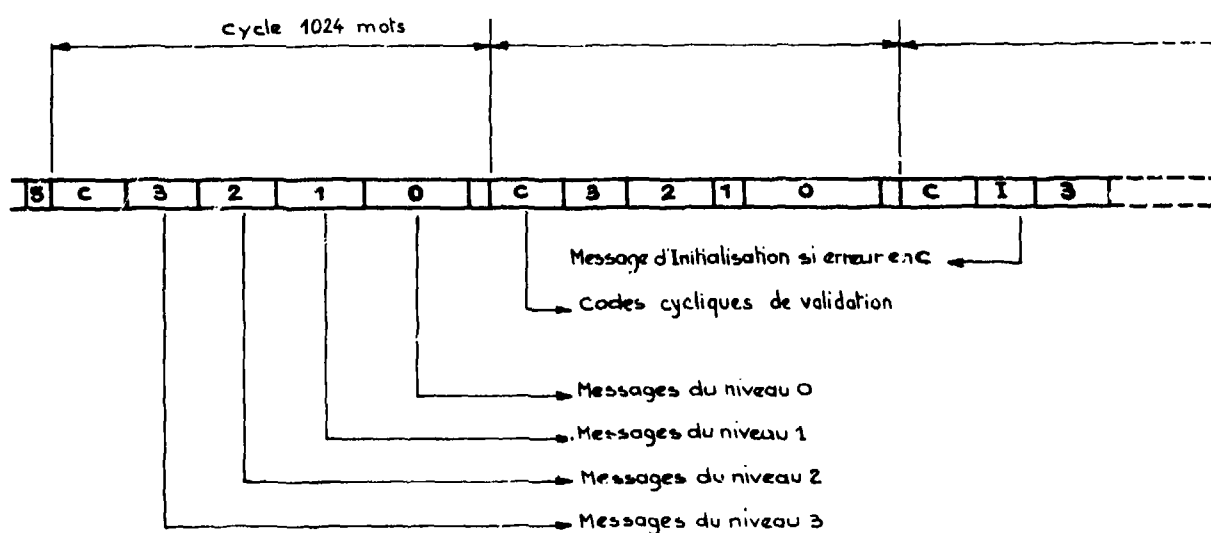


Figure 5 — STRUCTURE DE LA TRAME DYNAMIQUE

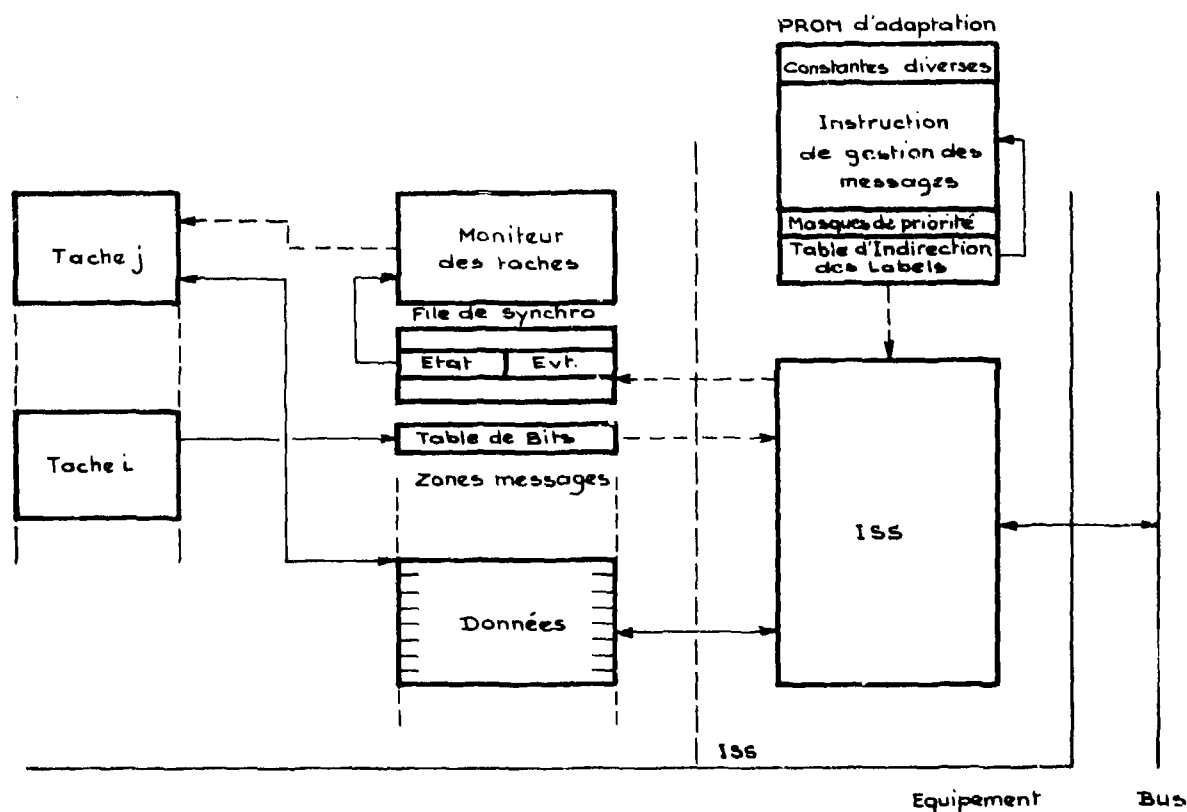
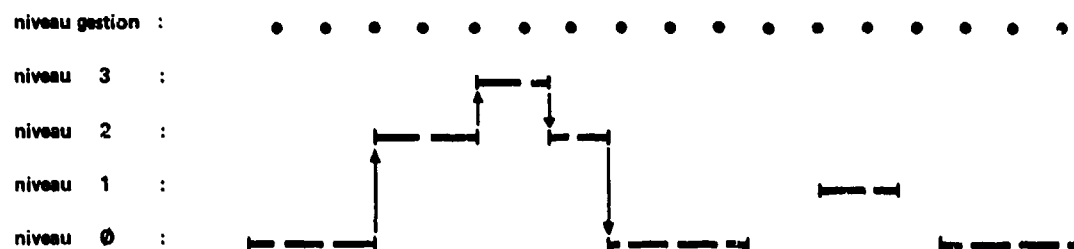
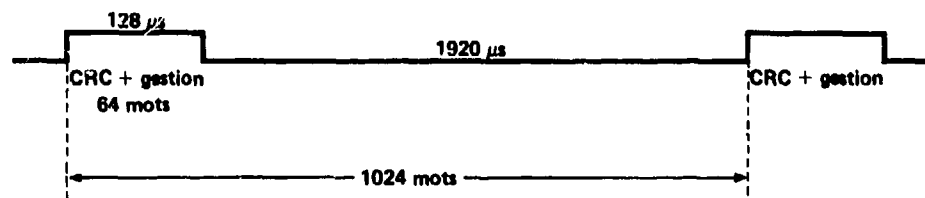


Figure 6 — SYNCHRONISATION ENTRE L'ISS ET L'EQUIPEMENT



- Sur un niveau, le bus est attribué aux ISS par ordre d'adresses croissantes
- L'interruption en faveur des niveaux prioritaires est prise en charge derrière un cycle de gestion (pointillé)
Latence < 2 ms

Figure 7 — TRAME

DISCUSSIONS
SESSION V

REFERENCE NO. OF PAPER: V-20

DISCUSSOR'S NAME: Dr. G. H. Hunt, AVP Member

AUTHOR'S NAME: A. D. Stern

COMMENT: It is an implicit assumption in your paper that the occurrences of failure in the different LRU's are completely uncorrelated. It seems to me that there may be some mechanisms for failure and degradation, particularly those associated with variations in environmental conditions, which are common to many LRU's and which could correlate to some measure the occurrence of failure. Has the author been able to satisfy himself that this is not so?

AUTHOR'S REPLY: I agree with the comment entirely. However, common failure modes should be designed out of the system, otherwise why bother with redundancy. The stage-state method presented is typical to other reliability analysis methods in that it assumes randomness of failure occurrence. This is sufficient for architecture reliability trade studies and comparisons for which most methods are intended.

There is some work going on in reliability which addresses different failure rates for recognition of the fact that most failures occur during near "turn-on" time, and that failure rates for a given device may change if it is in some standby mode.

REFERENCE NO. OF PAPER: V-22

DISCUSSOR'S NAME: K. A. Helps, Smith Industries

AUTHOR'S NAME: R. L. Schwartz

COMMENT: In making your analysis of the SIFT system depend on a sequence of proofs of the equivalence in certain respects of one model to the next in the sequence from I/O model to Pascal program (figure 5), have you made any estimate of the likelihood of correctness of the proofs necessary to match the extremely low probability of system failure required (1 in 10^{10} per hour), and is there not a need to have (dissimilar) redundancy of proofs since human argument is fallible even when supported by mechanical aids such as theorem provers? (Although a proof is either correct or incorrect, confidence in a proof is generally not 100%.)

AUTHOR'S REPLY: You are indeed correct that the validity of the overall SIFT verification efforts depends in part on the soundness of the mechanical theorem prover employed. The answer to this is not however to replicate the proof process (with perhaps some sort of majority vote??) or to use a probabilistic analysis of the proof validity.

The role of any verification attempt is to increase confidence in a system. That a machine-generated axiomatic proof is correct merely means that the truth of the proposition follows from a given set of axioms and rules of deduction. This by itself is not difficult to check, either by machine or by a human reader. This is not the problem area. Determining that the proposition (or specification) actually expresses a property sufficient to ensure the behavior you intend is the more difficult problem. That the I/O model, the highest level description of SIFT, expresses the intended system function in the end must be determined by inspection.

I do not believe, however, that this is the weakest link in the argument of SIFT's reliability. To me, the weakest argument concerns the reliability assumptions for the underlying hardware which were made in order to employ a Markov reliability analysis. Assumptions such as the statistical independence of faults in various processors and that their occurrence satisfies a negative exponential distribution appear to me more bothersome. It is here that further work should be done. Eventually, I also expect that mechanical theorem provers will be shown to be consistent with a specification--pushing the problem back one more level.

REFERENCE NO. OF PAPER: V-22

DISCUSSOR'S NAME: J. H. Saltzer, MIT, USA

AUTHOR'S NAME: R. Schwartz

COMMENT: Is it your experience that the process of systematic design and verification is actually uncovering design errors?

AUTHOR'S REPLY: Yes, I believe the formal specification and verification process has been quite useful in uncovering incomplete and flawed design decisions. Probably the flaw with the widest significance is that clock synchronization cannot be guaranteed to withstand a single point failure using triplication and majority voting. In response to this, a new "interactive consistency" algorithm was developed. The problem, solution, and its proof can be found in an article by Pease, Shostak, and Lamport in the Journal of the ACM, April 1980. There are many other instances involving neglected and necessary constraints on the schedule table, etc. In general, formal specification and verification forces every possible state of the system to be considered and thus is one of the (if not the) most systematic analyses possible.

REFERENCE NO. OF PAPER: V-22

DISCUSSOR'S NAME: Schwartz, SRI, USA

AUTHOR'S NAME: Enslow (Livesey, Presenter)

COMMENT: Is your proof effort primarily oriented towards design verification or towards implementation verification? For example, if I accepted your figure of 10^{-10} for fault independence, but asserted that two processes would deadlock with a probability of 10^{-9} , does your work address that?

AUTHOR'S REPLY: The proof effort extends from the highest level design specifications down through and including the Pascal implementation. The highest level model specifies that the system must continue to apply the correct task function on the correct input values. Any implementation claiming to implement SIFT must satisfy this, therefore, lower level design decisions which result in deadlock, for example, will thus not satisfy this requirement.

I might comment that because we are verifying the validity of employed fault-tolerance algorithms within our design verification, the lowest level proof of Pascal implementation is rather trivial.

REFERENCE NO. OF PAPER: V-23

DISCUSSOR'S NAME: Jim McCuen, Hughes Aircraft, USA

AUTHOR'S NAME: M. Szlachta

COMMENT: Why did you select to use the MIL-STD-1553 Bus for the intertie for the computers?

AUTHOR'S REPLY: We had to chose between things available at the moment. The 1553 is supported by many people. We are not fully satisfied with it. Probably because we are misusing the bus--we are using it as a processor link and had to change it a little.

REFERENCE NO. OF PAPER: V-23

DISCUSSOR'S NAME: G. Scotti, SELENIA

AUTHOR'S NAME: J. Szlachta

COMMENT: In normal operation only one computer has the possibility to access memory and I/O, while the second CPU is maintained hot. If the crosscoupler disconnects the nonactive processor from the memory and the I/O, how is it possible to compare the correct operations between the two processors in absence of correct data in the second CPU?

AUTHOR'S REPLY: The crosscoupler prevents only the output and not the input. That means that the output of the active CPU is written into both memory blocks. The line in figure 3 indicates the active data flow only. The additional data flow, for example, error checking is not shown.

REFERENCE NO. OF PAPER: V-23

DISCUSSOR'S NAME: Horst Kister, VDO

AUTHOR'S NAME: Szlachta

COMMENT: What does "synchronization" in this case mean?

AUTHOR'S REPLY: By synchronization we understand the periodic exchange of messages between the distributed copies of a systems-process. This exchange serves to secure the consistency of a subset of the global data space.

REFERENCE NO. OF PAPER: V-24

DISCUSSOR'S NAME: Jim McCuen, Hughes Aircraft Co., USA

AUTHOR'S NAME: M. Meraud

COMMENT: Have you built a fiber optic bus (100M to 300M) with 32 taps/drops? If so, have you operated it successfully?

AUTHOR'S REPLY: Not yet. This was a study. The ISS is under design and we will have it in 1982. As for the optical part, it is the other society which is in charge of it. We expect it to have it ready by mid-1982. And a prototype should be ready by the end of that year.

PROTOCOL LEVEL MODULES - FOR COST EFFECTIVE STANDARD COMPUTER COMMUNICATION

Gyvind Hvinden, Yngvar Lundh, Øystein Sandholt

Norwegian Defence Research Establishment, P O Box 25 - N-2007 Kjeller, Norway

SUMMARY

A set of microcomputer modules for implementation of network front-end, gateway and specialized host computers are being developed. A highly modular design approach is taken. One or more of these protocol modules may be interconnected to constitute the units referred to. A "library" of tested hardware sub modules is established, new modules may quickly be developed using these sub modules. A framework for unified protocol implementation and protocol interconnection is defined. This includes a real time operating system kernel with functions for buffer management, timing, pseudo parallel process execution and process communication.

1 INTRODUCTION

An experimental distributed computer system based on local networking is under development at the Norwegian Defence Research Establishment (NDRE). This effort includes development and investigation of different local nets, gateways between these nets and existing long haul nets, host computer network interfaces, host computer network software and specialized host computers. This paper concentrates on techniques for implementation of "Network Front-Ends" (NFE), gateways and specialized host computers based on microprocessor technology.

Local network technology is rapidly developing and various network types are needed in our experiments. A network architecture that permits experimentation with different nets without having to redesign host computer network interfaces is highly desirable. The front-end technique hides network specific details and host interfaces may be standardized independently of the underlying net. This technique has both advantages and disadvantages compared to "non-intelligent" hardware interfaces where the host carries out all protocol software. Application independent protocols that are commonly used by all hosts are protocols well suited for front-end implementation. The implementation effort for these protocols may then be reduced to one processor type. Carrying out protocol functions by a front-end may offload an expensive host computer significantly, thus providing more host capacity to the users.

On the other hand a front-end may be a throughput bottleneck for hosts with high performance network requirements. Increased packet delays may also occur.

For our current applications a front-end technique based on standard microprocessor technology is adequate in terms of throughput and delay. These applications are narrow band digital voice terminals, terminal interface units and general host (mini-) computer networking (remote terminal access and file transfer).

2 ARCHITECTURE AND MODULARIZATION OF NETWORK FRONT-END GATEWAYS AND HOST COMPUTERS

The ISO reference model for "Open Systems Interconnection" (ISO TC97/SC 16, 1979) provides the framework for protocol implementation. Protocols are hierarchically layered and communicates internally only with protocols above and underneath it in the hierarchy. The hardware and software base for protocol implementation should preferably support this architecture.

The seven layer ISO hierarchy is divided into three main parts, network layers (1-3), transport control layer (4) and application oriented layers (5-7). The transport control layer and layers above are network independent. Layers 1-3 (physical link, link access and network) are used by all hosts, while different transport protocols may exist within a network. Choice of transport control protocols depends on network community (ARPA, X.25, ...) and application (file transfer, speech, ...) which may have different requirements with respect to reliability and delay. The US-DOD ARPA Transport Control Protocol (ARPA IEN-129, 1980) is a transport control protocol for extremely reliable communication in all imaginable conditions. Such a protocol may be "overkill" for local communication on fast, almost error free nets, and a solution with co-existing transport protocols may be advantageous. The transport protocol shall provide reliable service to the users of it, that means a reliable path from the transport protocol to its users must exist.

The network layers are used by all hosts and gateways and are therefore obviously well suited for front-end implementation. The transport protocol may also advantageously be implemented in the front-end if a reliable host-front end interface is obtainable. Transport control protocols are often very complex and resource demanding both to implement and execute.

Protocols above the transport layer are inherently host specific and in general not suitable for front-end implementation.

A host computer may be connected to the front-end by various types of interfaces, parallel or serial. Parallel interfaces are usually the best solution, they are fast, reliable and they represent little host processing overhead, especially if they are DMA driven. Serial interfaces provides a "clean" interface and permit greater distance between host and front-end than high speed parallel interfaces. A serial interface is usually used with a link access protocol that provides reliable flow controlled service.

We may now conclude that several combinations of host interface and protocol level for interfacing are functionally equivalent and of current interest. Which combination to select depends on host type, network type, application and economy. No method is obviously best for all purposes. It means that a flexible front-end design that permits use of different nets, different host interfaces and a variable number of protocol levels is beneficial.

According to the layered structure in the ISO reference model a modular layered front-end design approach is sensible. We decided to implement one or more protocols on dedicated boards and interconnect these boards to constitute the units to be developed. These boards are referred to as "Protocol Level Modules" (PLM) or just "protocol modules".

A protocol module has at least two interfaces, upper and lower. When more than one protocol module is needed to constitute a unit, the protocol modules in the unit communicate internally by a "Inter Level Interface" (ILI). A protocol module that interfaces to a host has a "Host Specific Interface" (HSI). The ILI is standardized while the HSI may differ depending on host type. The lowest level protocol module has a Network Specific Interface (NSI) and protocol modules carrying out applications may have an "Application Specific Interface" (ASI). Since protocol modules are single board units, several types have to be developed to cover network and interface combinations. This concept is not practical if extremely many such combinations were needed. A multi-board processor design may then be a better solution. The following examples will show how such modules may be used in front-ends as well as in gateway and specialized host computers. Figure 2.1 shows 3 front-end configurations which we are implementing.

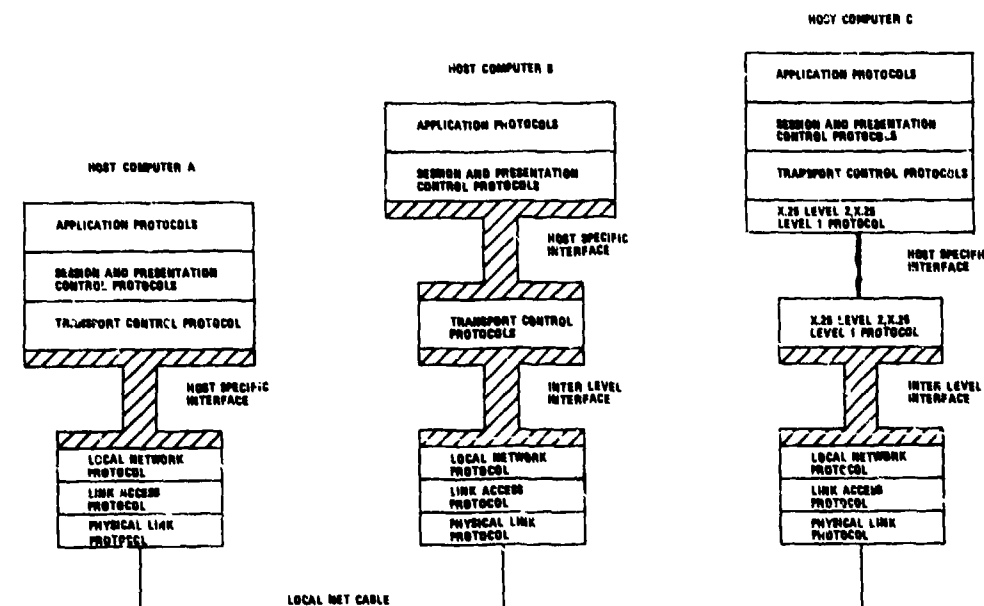


Figure 2.1 Host network front-end interface examples

The low level protocol module contains the local network protocols used by all network stations. Host "A" is connected to the front-end by the HSI at the network level. The HSI is a parallel interface in this case. Host "B" is connected by a similar interface at the transport protocol level. Two protocol modules are used in this front-end, interconnected by the ILI. Host "C" is connected at the network level, a X.25 link access and physical link constitutes the HSI here. Scope of X.25 is limited to host front-end access in this example.

In Figure 2.2 protocol modules are used in two gateway configurations. The gateway between A and C are constituted of protocol modules throughout, one network module for each net with the gateway protocol in a module in between. Between similar nets this design is simple and straight forward. Gateways between very different nets may be more complicated, and more powerful computers may then be needed. A (e.g. mini-) computer already interfaced to an existing net may be converted to a gateway between two nets by interfacing it to a front-end, connected to the new net. The gateway between network B (long haul store forward net) and network C exemplifies this design.

Specialized host computers are machines that are dedicated to special purposes. Typical examples are terminal interface units for remote terminal access to host computers and speech terminals for digitized, packetized speech. The number needed of such units may be relatively large and cost-effective solutions are important. Figure 2.3 shows these two hosts built completely of protocol modules. The speech host has interfaces to a speech digitizer (vocoder) and key-pad/display for connection control and status, the terminal host has standard RS-232 interfaces.

We have now discussed a system architecture based on protocol modules with several examples of use, both potential and currently implemented. Before going further into design details, two issues regarding this architecture will be addressed.

Firstly, the number of protocol modules "stacked" to constitute a unit should be kept low to keep both cost and packet (traffic) delays low. The ILI design is particularly important here, and delays may be kept very low if microprocessors with block transfer capability or "Direct Memory Access" (DMA) are used.

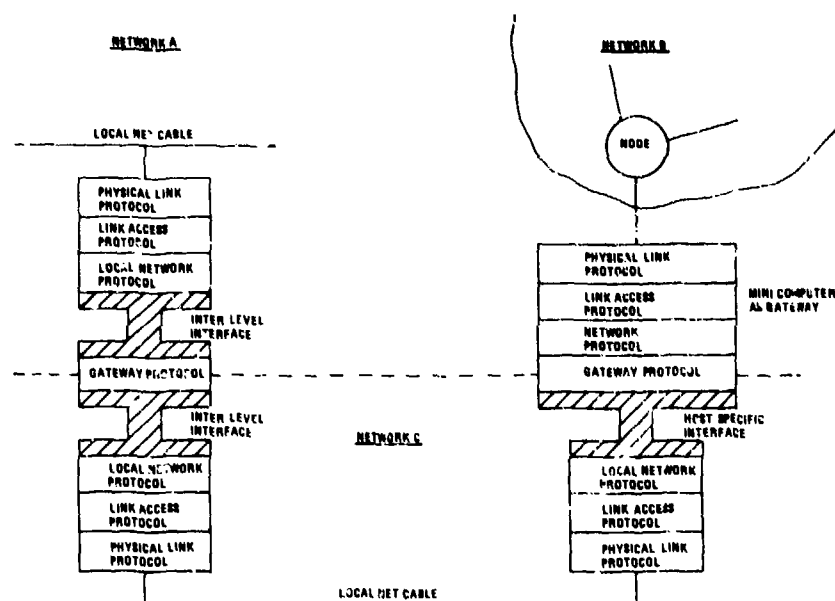


Figure 2.2 Use of protocol modules in gateways, an example

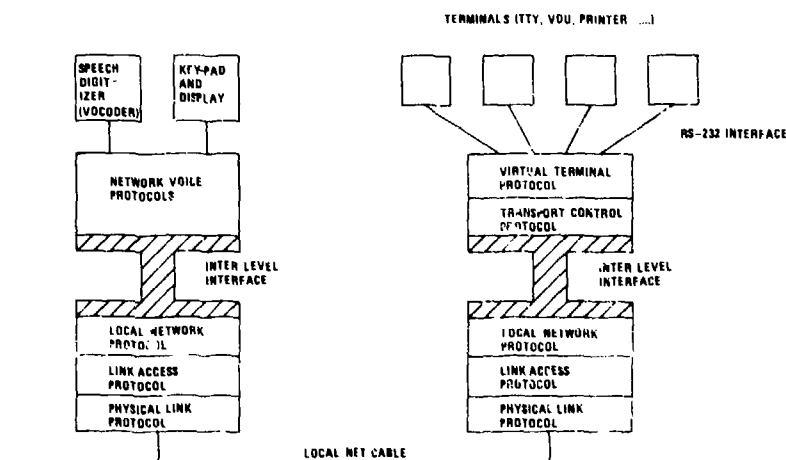


Figure 2.3 Protocol module based specialized host computers

Secondly, using more than one protocol module permits "tailored" implementation of protocols. Low level (e.g. link) and application protocols (e.g. terminal controller) often need very fast and efficient interrupt handling, while medium level protocols (e.g. network and transport control) beneficially are implemented under an operating system kernel. Meeting both these requirements with one processor means compromises and more complex software.

3 PROTOCOL MODULE HARDWARE INTERCONNECTION AND ARCHITECTURE

Based on the system architecture discussed a set of protocol modules have been developed. The following main requirements were set as design goals:

- Protocol modules are single board microcomputers
- It must be possible to use different microprocessors as protocol module CPU, both "8 and 16 biters"
- A standard inter module interface for fast, efficient inter module packet exchange should be defined. This interface must not lock CPU's tightly together for long times during transfer
- Flexible combination of RAM and EPROM memory should be possible
- The concept should allow exploitation of new powerful microprocessors and associated circuitry as it becomes available.
- Initial series of protocol modules should serve as experimental tools

A standard inter module interconnection had to be defined. This interface is a very important part of the system, and several solutions were analyzed before a decision was made. The interface must be parallel, in order to be fast and simple. The data transfer path must be 8 bits wide to accommodate both 8 and 16 bit

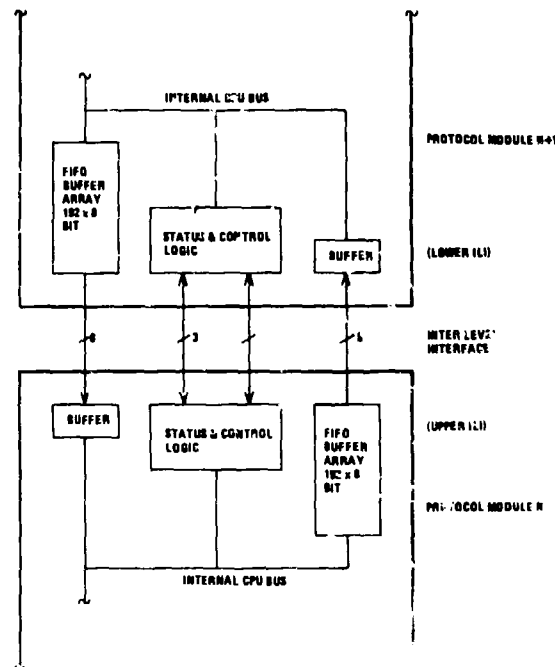


Figure 3.1 Protocol module "Inter-processor" block diagram

processors. It must be fast enough to handle DMA data rates (1 Mbyte/s) and asynchronous so that fast and slow processors may communicate without tying each other up. The interface must hide internal memory organization. That means "word" or "byte" memory organization are unacceptable. A design based on fifo circuits was selected. Packets of arbitrary length are transferred as one or more segments under software control through this interface, read and write operations are separated and are completely asynchronous. The fifo array capacity is 192 bytes. Figure 3.1 shows two protocol modules with this interface, named ILI.

The current implementation are based on 128 x 8 bit fifo chips and LS-TTL control and status circuitry. 21 chips are used in the present design and read/write state is occupied for complete upper and lower ILI. Chip count may be reduced considerably in future designs. 128 x 8 bit fifo chips complete with control, status and bus interface circuitry will be available from microprocessor manufacturers in 1981. Two 40 pin chips can then replace all of the present ones. These two interfaces will not be directly compatible.

Hardware resources needed by the protocols are CPU and memory. Any type of CPU may be used with this concept. The "A-CPU" has been used in the present design. A timer/timer circuit is needed in many protocols and it includes a standard device. The CPU must be able to start when power is switched on, initialize itself, become synchronized with real time, and before useful processing may take place. The program may be partly or completely in EPROM. A flexible memory structure is needed, memory should be organized in increments. RAM and EPROM requirements are difficult to predict and pin compatible RAM/EPROM are therefore used. Completely debugged programs (if such exist) can be permanently stored in EPROM. During development it is highly useful to download the program into RAM, only one bootstrap EPROM is then needed. It is expected to be useful to use a final design for operational use, to have protocol programs in EPROM.

Four protocol modules have been designed. Figure 3.1 shows their hardware architecture.

All protocol modules on Figure 3.1 contain the CPU and associated reset, buffer and clock circuitry and the first 192 kbytes of memory. "Universal" has both upper and lower ILI and 64 kbytes of memory. "HDLC" has 32 kbytes of memory, upper and lower ILI, a multi-protocol link level controller chip (Z80A-SIO) with X.25.1, HDLC, EIA-422, and frame relay, bit rate generating circuitry and a 16 bit LED display for debug purposes. The "VOC" board may be located on the top and bottom of a layered protocol module package since it has upper and lower ILI. "TA" contains an interface to a narrow band digital voice vocoder, lower ILI and 16 kbytes of memory. A subset of the external I/O bus is made accessible for I/O devices not practical to locate on the "VOC" board (display and keypad). "RING-NET" contains an interface to a commercially available 16 bit ring-net controller, 16 kbytes of memory and upper ILI. It contains two dual port buffer memories, 16 kbytes each, for simultaneous 10 Mbit/s ring send and receive operation.

Several other modules are planned with various host and application specific upper interfaces (e.g. standard IBM-4380, PDA, and other network interfaces (e.g. Ethernet).

A 40 pin chip will increase performance several times for some of these modules, such a device should be incorporated when redesign is complete. Data packet transfer between memory and ILI, and memory-memory transfers now consumes significant parts of available CPU cycles. A Z80A-DMA performs this 5 times faster than the Z80A-DMA instructions.

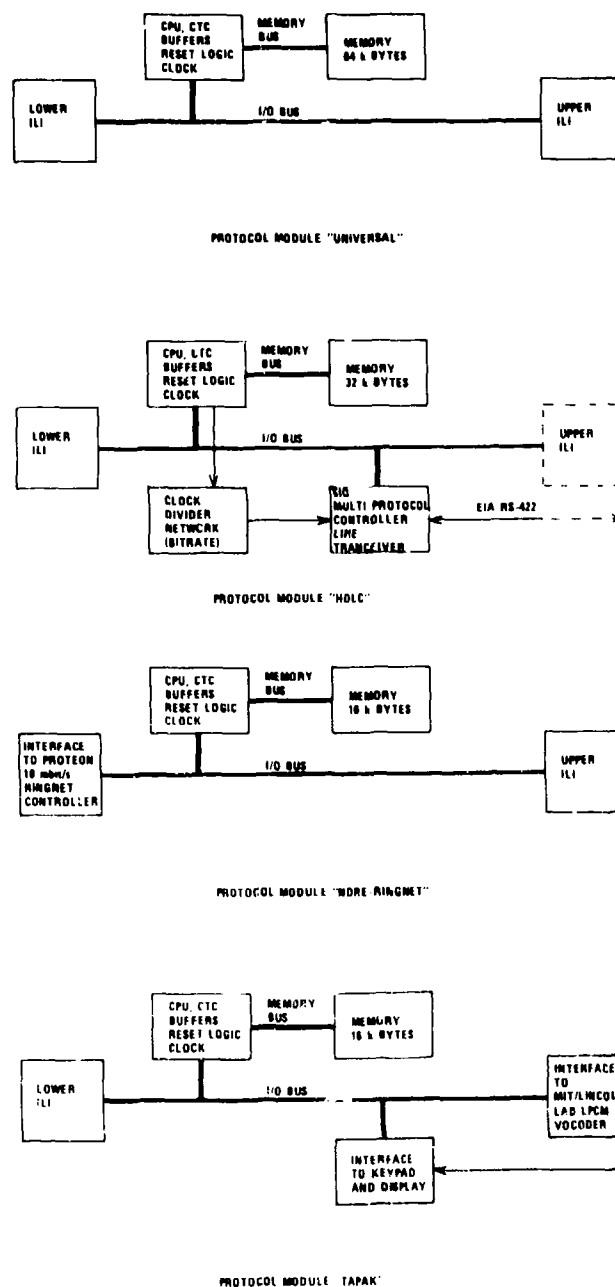


Figure 3.2 Protocol module block diagram

4 THE HARDWARE MODULE LIBRARY

An interactive layout system was used for the pc-board artwork design. The protocol module family has a modular architecture where certain sub-modules are used on many boards in the form of "library elements" in the layout system. Such submodules are, e.g. CPU and associated circuitry, upper and lower ILL and memory, see Figure 4.1. Figure 4.1 shows segment layout of the 4 boards developed.

Sub-modules are located on the same place on all protocol modules. A new board is designed by picking sub-modules from the library and placing them in an aggregate layout. Sub-modules which have been used before reduces development time of new modules substantially.

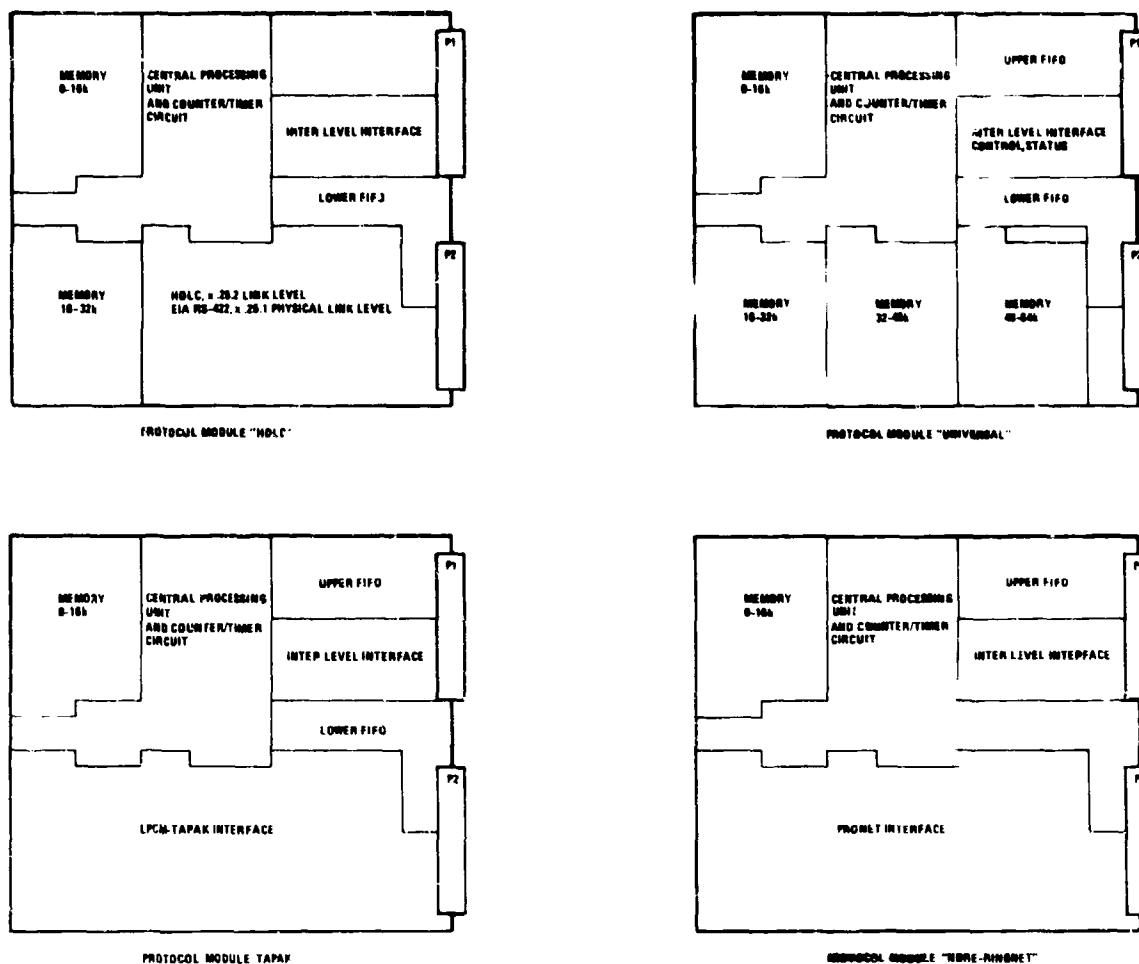


Figure 4.1 Protocol module segment layout

5 PROTOCOL MODULE SOFTWARE ARCHITECTURE

A unified software framework has been developed for structured and flexible protocol module interconnection. This framework supports communication between various software modules within a unit constituted of more than one protocol module. Such software modules are communication protocols and distributed functions for network debugging, network experiments and network maintenance. A typical example of protocol module and protocol interconnection structure is shown on Figure 5.1.

The host computer has various protocols that uses different protocols in the front-end package. The lines between the protocols symbolizes logical communication paths between them. We have defined a framework based on this structure. The abstract term "logical channel" is central in this framework. A logical channel is a one way communication path between software modules within one protocol module or between software modules in protocol modules that are neighbours in a hierarchy of layered protocol modules. Messages of finite length are exchanged on these channels and a rule for flow control is defined. A pool of logical channels is defined, this pool is divided into two main blocks, internal and external channels.

The external block is divided into two blocks, upward and downward channels. Figure 5.2 shows logical channel allocation.

As protocol development continues protocols will be assigned fixed external logical channel numbers, but this is not part of the framework.

The logical protocol for multiplexed message exchange is the high level part of the ILI and the HSI previously described. The low level part of these standards are the interface hardware and associated software drivers. The logical part of the protocol are independent of the hardware and driver part, which are subject to changes and new implementations.

We have defined two versions of the logical protocol, full and simplified ILI/HSI. Both depend on 100% reliable interface hardware and driver service. The difference is the flow control scheme used. The full

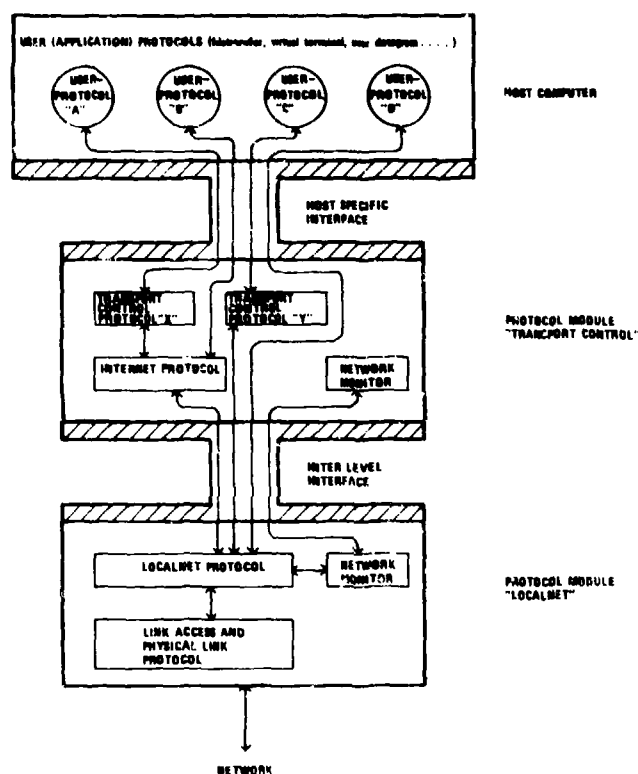


Figure 5.1 Typical protocol layering and interconnection

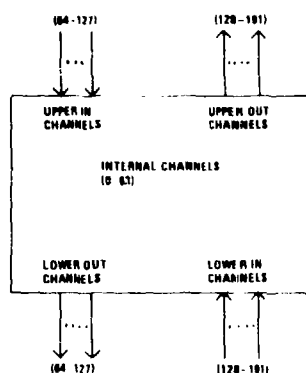


Figure 5.2 Logical channel block allocation

simplified version does not utilize these "request messages", and a sending "process" may overflow the receiving "process" so that the interface becomes temporarily blocked until the receiving "process" is able to accept a new message.

Message exchange can only take place on logical channels known in both protocol modules, messages sent on channels not allocated to specific "processes" end up in a "black hole" without any further notification to the sender "process".

6 STANDARD SOFTWARE MODULES

Certain parts of the protocol module software are similar on many protocol modules. I/O drivers are a typical example. Most protocols need functions for buffer memory management, message exchange and timing. Protocols above level 1 are implemented as real time programs and several protocols may share one CPU for execution. Some protocols are so complex that they must be divided into sub-modules that are to be treated as separate units for execution. A real time operating system that supports pseudo-parallel execution of such units or processes would simplify protocol implementation considerably. Such an operating system kernel imposes some processing overhead and is not well suited for protocol implementations with extreme response requirements (e.g. link level).

creation, buffer management, process communication and process scheduling. The process communication system is the logical channel system previously described. It supports both internal and external logical channel communication. This package is constituted of an inner kernel and a number of kernel processes that executes concurrently with processes carrying out the communication protocols. The ILI/HSI are implemented as kernel processes which have priority over protocol processes.

The kernel is now being used in an APPA TCP implementation effort which will need all kernel functions. The kernel design is not finished and frozen, we expect to modify it to new needs and to modify it for new processors. It is written in the Zilog PLZ/SYS system implementation language (Snook, T 1979) which allows migration to 16 bit processors with little conversion effort.

7 CONCLUSION

A system has been described for "host independent" implementation of communication protocols. References have been made to a preliminary design of such modules for experimental purposes. We believe that "network front-ending" using such a modular approach may have merit in future computer communications. Some of the reasons are: Maintenance and further development of the front-ended protocols becomes independent of the various host computer types. The work load on the host can be reduced substantially, and could be important for economy. Practical, application oriented implementations may exploit the expected further development of more powerful circuit technology, both microprocessors and more specialized circuits.

The most obvious limitations of "front-ending" are associated with the delay of packets travelling through the front-end. Further study of details of these limitations are under way. These investigations will establish quantitative factors for throughput, delay - and on the other hand circuit and program performance requirements for various situations. Certain applications with extreme performance requirements will probably still be better served by utilizing host computer capacity for the protocol logic - in the conventional manner.

References

ARPA IEN-129, 1980, "DOU Standard Transmission Control Protocol", Arpa RFC:761 IEN:129, Prepared for Darpa by Information Sciences Institute, University of Southern California

ISO TC97/SC 16, 1979, "Reference model of Open Systems Interconnection", ISO/TC97/SC 16 Working Document 227

Hvinden O, 1981, "The Paradis Kernel Software Package", FFI/NOTAT-81/7035, Norwegian Defence Research Establishment

Snook T, Bass C, Roberts I, Nahapetien A, Fay F, 1978, "Report on the Programming Language PLZ/SYS, Springer Verlag

LES STRATÉGIES DE RETRANSMISSION POUR LE CONTRÔLE D'ERREUR DANS LES PROTOCOLES DE TRANSFERT DE DONNÉES

GUY JUANOLE

LABORATOIRE d'AUTOMATIQUE
et d'Analyse des Systèmes du C.N.R.S.
7, avenue du Colonel Roche
31400 TOULOUSE - FRANCE

RESUME

Ce papier consiste en : d'une part, la présentation générale du transfert de données (paquets) à travers une ligne de transmission, compte tenu d'un contrôle d'erreur basé sur la détection d'erreur et la retransmission après détection d'erreur ; d'autre part, la définition et la présentation des différentes stratégies de retransmission.

La présentation générale du transfert de données est basée sur un modèle hiérarchisé à plusieurs niveaux où chaque niveau utilise les services du niveau inférieur.

Cette approche, essentielle pour une bonne visualisation des différentes fonctions nécessaires pour ce transfert, permet, en outre, de bien distinguer deux niveaux dans le contrôle d'erreur :

- un niveau supérieur relatif à un contrôle sur l'arrivée des paquets (mise en oeuvre de mécanismes de numérotation des paquets, de réponses aux paquets numérotés et de retransmission des paquets numérotés non acquittés),
- un niveau inférieur relatif à un contrôle sur le contenu des paquets numérotés et des réponses (mise en oeuvre de codes détecteurs d'erreurs).

La considération de ces deux niveaux est essentielle pour définir clairement et précisément les différentes stratégies de retransmission. Nous définissons deux classes de stratégies : la classe 1 où la retransmission est due uniquement à une temporisation qui est implémentée dans le niveau supérieur ; la classe 2 où la retransmission est également mise en oeuvre suite aux erreurs détectées par le niveau inférieur, que ce dernier signale au niveau supérieur.

Dans chaque classe, nous définissons les différentes stratégies qui résultent des différentes modalités possibles pour :

- a) l'envoi des paquets numérotés par la source de ces paquets numérotés,
- b) l'acceptation des paquets numérotés par le puits de ces paquets numérotés,
- c) la manière dont le puits accuse réception des paquets numérotés acceptés.

Cette présentation des différentes stratégies de retransmission nous apparaît comme une première étape essentielle avant d'effectuer leur modélisation formelle en vue d'une part d'une vérification de leur validité logique et d'autre part d'une implémentation.

Liste des symboles utilisés

P_{xi} (P_{xj})	: Processus de niveau x ($x = 1, 2, 3, 4, 5$) dans le calculateur C_i (C_j)
TEMP	: TEMPorisation
MT	: Machine de Transmission
$[PQ]_i$: Paquet émis par P_{5i} et destiné à P_{5j}
$[NPQ]_i$: $[PQ]_i$ Numéroté émis par P_{4i} et destiné à P_{4j}
$[RP]_j$: Réponse émise par P_{4j} et destinée à P_{4i}
$[RP(ACC)]_j$: $[RP]_j$ qui est un ACCusé de réception
$[RP(RET)]_j$: $[RP]_j$ qui est une demande de RETransmission
$[RNPQ]_i$: $[NPQ]_i$ Redondant émis par P_{3i} et destiné à P_{3j}
$[RRP]_j$: $[RP]_j$ Redondant émis par P_{3j} et destiné à P_{3i}
$[T]_i$: Trame série émise par P_{2i} et destinée à P_{2j}
$[T]_i$: Trame série émise par P_{2i} et destinée à P_{2i}
$[SE]_i$: Signal d'Erreur émis par P_{3i} et destiné à P_{4i}
$[SE]_j$: Signal d'Erreur émis par P_{3j} et destiné à P_{4j}
$[PQ]_i, [RP]_j, \dots$: pluriel de $[PQ]_i, [RP]_j, \dots$

INTRODUCTION

La fiabilité des applications distribuées (qui se multiplient actuellement compte tenu du développement des systèmes de calculateurs géographiquement distribués) dépend, en particulier, de la fiabilité du transfert des données au moyen des lignes de transmission, ce qui donne, donc, toute son importance au contrôle d'erreur appliqué à ce transfert. D'une manière générale, ce contrôle d'erreur est basé sur les principes suivants : détection d'erreurs et retransmission après détection d'erreurs.

Le système, qui sert de support à notre analyse, consiste en deux calculateurs C_i et C_j , situés dans deux sites distants et connectés au moyen d'une liaison point à point. Nous appelons, paquet, la structure de donnée à être échangée entre C_i et C_j à travers la ligne de transmission. Nous considérons uniquement le cas du transfert unidirectionnel de paquets : en effet, les principes des stratégies de retransmission de ce type de transfert se retrouvent dans les transferts bidirectionnels.

L'analyse effectuée comprend deux parties : dans une première partie, nous représentons le transfert de données par un modèle hiérarchisé ce qui nous permet de visualiser les deux niveaux du contrôle d'erreur et les classes de stratégies de retransmission; dans une deuxième partie, nous définissons les différentes stratégies de retransmission.

1. MODEL HIERARCHISE DU TRANSFERT DE DONNEES

1.1 Présentation générale du système considéré

Le système considéré est représenté sur la figure 1 :

- P_{5i} représente à la fois les processus d'applications de C_i et les processus nécessaires pour transformer les structures de données de ces applications en paquets à envoyer vers C_j ; P_{5j} représente à la fois les processus d'application de C_j et les processus nécessaires pour transformer les paquets reçus en structures de données significatives pour ces applications,
- les processus des niveaux 4,3,2 et 1 mettent en oeuvre les principales fonctions nécessaires au transfert de données (séquence de paquets) depuis C_i vers C_j , à travers une ligne de transmission (niveau 0).

La décomposition multiniveaux amène à distinguer deux types de protocoles : des protocoles relatifs à la coopération entre processus distants qui sont dans un même niveau que nous les appelons des "protocoles de niveau" ; des protocoles relatifs à la coopération entre des processus dans deux niveaux adjacents d'un calculateur que nous appelons des "protocoles entre deux niveaux" (un "protocole entre deux niveaux" est caractérisé par un ensemble de primitives relatives au service qu'un niveau demande au niveau immédiatement inférieur). Nous ne considérons pas ici les "protocoles entre deux niveaux."

Nous décrivons maintenant les principales fonctions des protocoles des niveaux 4,3,2 et 1 en insistant plus particulièrement sur les niveaux 4 et 3 qui mettent en oeuvre le contrôle d'erreur.

1.2 Les protocoles des niveaux 4,3,2 et 1

1.2.1 Protocole du niveau 4

Ce protocole a pour but de contrôler si les $[PQ]_i$ arrivent à leur destination (les niveaux inférieurs peuvent perdre des informations qui y transitent). Afin de réaliser ce contrôle, on a donc les échanges suivants au moyen du niveau 3 : des $[NPQ]_i$ de P_{4i} vers P_{4j} et des $[RP]_j$ de P_{4j} vers P_{4i} . Compte tenu de ce contrôle les fonctions de P_{4i} et P_{4j} sont :

- P_{4i} - Il numérote les $[PQ]_i$ que P_{5i} lui demande d'envoyer (mise en forme des $[NPQ]_i$), transmet ces $[NPQ]_i$ à P_{3i} et attend des $[RP]_i(ACC)_i$ afin de pouvoir prendre de nouveaux $[PQ]_i$ et donc transmettre de nouveaux $[NPQ]_i$; les $[NPQ]_i$, dont les $[RP]_i(ACC)_i$ ne sont pas reçus, sont retransmis (stratégie de retransmission).

D'une manière générale, la retransmission intervient à la fin d'une TEMP (METCALFE,73) ou quand P_{4i} reçoit des $[RP]_i(RET)_i$ ou des $[SE]_i$ (nous allons voir dans l'analyse du niveau 3 que P_{3i} a la capacité de détecter des erreurs et peut donc signaler ces erreurs à P_{4i}). Notons également que l'on peut également envisager la retransmission quand P_{4i} reçoit des $[RP]_i(ACC)_i$ non attendus (c'est-à-dire concernant des $[NPQ]_i$ non envoyés).

- P_{4j} - Il accepte ou rejette un $[NPQ]_i$ reçu, suivant que son numéro est ou n'est pas un numéro attendu et envoie des $[RP]_i(ACC)_i$ (le fait que P_{4i} envoie un $[RP]_i(ACC)_i$ quand il rejette un $[NPQ]_i$ peut paraître surprenant aux personnes non familières avec les stratégies de retransmission ; nous expliquerons ceci au paragraphe II). P_{4j} peut également recevoir des $[SE]_j$ (P_{3j} , comme P_{3i} , a la capacité de détecter des erreurs) et dans cette hypothèse, P_{4j} envoie des $[RP]_j(RET)_j$.

En ce qui concerne P_{4j} , il doit encore assurer une autre fonction : le maintien, vis à vis du niveau 5, de la séquence des $[PQ]_i$, c'est-à-dire P_{4j} doit réordonner les $[NPQ]_i$ acceptés (dans l'hypothèse où il peut les accepter dans le désordre) avant de pouvoir, après avoir enlevé le numéro, transmettre à P_{5j} les $[PQ]_i$ dans l'ordre où ceux-ci ont été transmis à P_{4i} par P_{5i} . Les numéros dans les $[NPQ]_i$ et les $[RP]_j$ constituent les "informations de service" du niveau 4.

1.2.2 Protocole du niveau 3

Ce protocole a pour but de contrôler si le contenu des $[NPQ]_i$ et des $[RP]_j$ n'est pas erroné (les niveaux inférieurs peuvent altérer les informations qui y transitent). Afin de réaliser ce contrôle, on a donc les échanges suivants au moyen du niveau 2 : des $[RNPQ]_i$ de P_{3i} à P_{3j} et des $[RRP]_j$ de P_{3j} à P_{3i} . Compte tenu de ce contrôle, les fonctions de P_{3i} et P_{3j} sont :

- P_{3i} (P_{3i}) ajoute un bloc de redondance à chaque $[NPQ]_i$ ($[RP]_j$) que P_{4i} (P_{4j}) lui demande d'envoyer et transmet donc, à chaque fois, un $[RNPQ]_i$ ($[RRP]_j$) à P_{2i} (P_{2j}),
- P_{3i} (P_{3j}) effectue le test du bloc de redondance associé à chaque $[RNPQ]_i$ ($[RRP]_j$) que lui transmet P_{2i} (P_{2j}) : si ce test est satisfaisant, P_{3i} (P_{3j}) enlève le bloc de redondance et transmet chaque $[RP]_i$ ($[NPQ]_j$) ainsi obtenu à P_{4i} (P_{4j}) ; si ce test n'est pas satisfaisant, P_{3i} (P_{3j}) ne transmet aucun $[RP]_i$ ($[NPQ]_j$) à P_{4i} (P_{4j}) mais, par contre, peut transmettre un $[SE]_i$ ($[SE]_j$) à P_{4i} (P_{4j}).

Les blocs de redondance sont les "informations de service" du niveau 3.

1.2.3 Protocole du niveau 2

Ce protocole assure la transmission d'informations sous forme série (bits). Dans ce but, on a les échanges suivants au moyen du niveau 1 : des $[T]_i$ de P_{2i} à P_{2j} et des $[T]_j$ de P_{2j} à P_{2i} . L'article de METCALFE, 73, décrit de manière exhaustive les fonctions que doivent réaliser P_{2i} et P_{2j} .

1.2.4 Protocole du niveau 1

Ce protocole permet l'échange de bits au moyen d'une ligne de transmission. L'ouvrage de [LUCKV,68] indique les fonctions que doivent assurer P_{1i} et P_{1j} .

1.3. Modèle considéré pour définir et présenter les stratégies de retransmission

Etant donné que le mécanisme de retransmission est élaboré à partir de P_{4i} (niveau 4) soit à la suite de situations d'erreur détectées par la TEMP dans P_{4i} soit à la suite de situations d'erreur détectées dans le niveau 3 et signalées au niveau 4, nous considérons donc le modèle à 3 niveaux représenté sur la figure 2 :

- le niveau 5 représente le niveau demandeur du service pour lequel le mécanisme de retransmission est nécessaire,
- le niveau 4 représente le niveau qui implémente ce mécanisme,
- la Machine de Transmission (MT), qui englobe les niveaux inférieurs, représente la machine globale utilisée par le niveau 4 pour la mise en oeuvre du mécanisme de retransmission.

1.4. Les deux grandes classes de stratégies de retransmission

Classe 1 : La retransmission résulte seulement de la TEMP dans P_{4i} . Dans cette classe, la MT ne transmet pas les $[SE_{s,i}]$ et $[SE_{s,j}]$ au niveau 4 et donc, en particulier, les $[RP_s(RET)]_j$ n'existent pas. En outre, P_{4i} ne tient pas compte des $[RP_s(ACC)]_j$ non attendus qu'il reçoit.

On peut dire que la classe 1 représente la classe des stratégies de retransmission avec le minimum d'actions causant la retransmission.

Classe 2 : Dans cette classe, la MT transmet les $[SE_{s,i}]$ et $[SE_{s,j}]$ au niveau 4 et celui-ci les prend en compte afin de déclencher le mécanisme de retransmission comme à la fin de la TEMP (la classe 2 englobe donc la classe 1).

Le but de la classe 2 est double : d'une part, prendre en compte l'intelligence du niveau 3 ; d'autre part, permettre, par rapport à la classe 1, un meilleur débit d'information en activant la retransmission.

Notons de plus que, du fait du signal $[SE_{s,i}]$, on doit en règle générale avoir deux types de $[RP]_j$, $[RP(ACC)]_j$ et $[RP(RET)]_j$ alors que seul le premier type existe dans la classe 1.

1.5. Hypothèses de définition des stratégies de retransmission

Nous considérons que :

- dans les conditions normales de fonctionnement (pas d'erreurs), P_{4j} envoie un $[RP(ACC)]_i$ à chaque $[NPQ]_i$ accepté ; nous ne considérons pas ici le cas où P_{4j} recevrait plusieurs $[NPQ]_{s,i}$ et enverrait ensuite un seul $[RP(ACC)]_i$ qui acquitterait cumulativement tous les $[NPQ]_{s,i}$ (comme dans le protocole HDLC en mode NRM avec les bits P/F (MACCHI,79)),
- une TEMP est associée, dans P_{4i} , à chaque $[NPQ]_i$ envoyé,
- la durée de la TEMP associée à chaque $[NPQ]_i$ envoyé, est telle que, un $[RP]_j$ à ce $[NPQ]_i$ est obtenu par P_{4j} avant la fin de la TEMP ou n'est pas obtenu (nous ne sommes pas concernés ici par des milieux de transmission induisant des retards qui pourraient faire arriver un $[RP]_j$ après la fin de la TEMP et donc durant l'opération de retransmission).

2. LES STRATEGIES DE RETRANSMISSION

2.1. Classe 1

2.1.1. Définition et propriétés

La première considération, afin de définir les différentes stratégies, concerne les deux modalités possibles pour l'envoi de $[NPQ]_i$ par P_{4i} :

- le mode 1, c'est-à-dire P_{4i} envoie seulement un $[NPQ]_i$ et ensuite attend le $[RP(ACC)]_j$ relatif à ce $[NPQ]_i$ afin de pouvoir envoyer le $[NPQ]_{s,i}$ suivant, ou la fin de la TEMP associée à ce $[NPQ]_i$ pour le renvoyer,
- le mode 2, c'est-à-dire P_{4i} envoie plusieurs $[NPQ]_i$ (nous considérons $(q+1)$ où q est un entier positif dont la valeur est une contrainte dans une implémentation) et ensuite attend "des $[RP(ACC)]_j$ ou la fin de la TEMP associée au premier des $(q+1)$ $[NPQ]_i$ " (cette partie de phrase entre guillemets est floue, c'est-à-dire : quel est l'ordre d'arrivée des $[RP(ACC)]_j$ pour qu'ils soient pris en compte et quelle action est entreprise quand un $[RP(ACC)]_j$ a été pris en compte ? Quelle action est entreprise à la fin de la TEMP indiquée ? Nous leverons le flou de cette phrase dans la suite de ce paragraphe en précisant les comportements possibles de P_{4j} ; notons que nous supposons qu'aucun $[RP]_j$ ne peut être reçu par P_{4i} avant la fin de l'envoi du $(q+1)^{\text{ème}}$ $[NPQ]_i$.

En utilisant la notion de $[NPQ]_i$ en transit ($[NPQ]_i$ envoyé par P_{4i} mais dont le $[RP(ACC)]_j$ n'a pas encore été obtenu par P_{4j}), on peut encore dire pour distinguer les modes 1 et 2 : dans le mode 1, il y a un $[NPQ]_i$ en transit ; dans le mode 2, il y a $(q+1)$ $[NPQ]_i$ en transit.

Les caractéristiques du mode 1 sont : P_{4j} nécessairement accepte les $[NPQ]_{s,i}$ en séquence ; après avoir accepté un $[NPQ]_{s,i}$, P_{4j} envoie immédiatement le $[RP(ACC)]_j$ relatif à ce $[NPQ]_{s,i}$.

La deuxième considération concerne dans l'hypothèse du mode 2, les deux politiques possibles d'acceptation de $[NPQ]_{s,i}$ par P_{4j} :

- la politique 1, c'est-à-dire P_{4j} accepte les $(q+1)$ $[NPQ]_{s,i}$ seulement en séquence (comme dans le mode 1),
- la politique 2, c'est-à-dire P_{4j} accepte les $(q+1)$ $[NPQ]_{s,i}$ dans n'importe quel ordre.

La politique 1 a les conséquences immédiates suivantes :

- si le premier des $(q+1)$ $[NPQ_s]_i$ n'est pas obtenu par P_{4j} , P_{4j} normalement rejettera les q $[NPQ_s]_i$ suivants,
- donc, compte tenu de ce comportement de P_{4j} , on peut maintenant préciser celui de P_{4i} (cf phrase floue) : P_{4i} attend seulement le $[RP(ACC)]_j$ relatif au premier des $(q+1)$ $[NPQ_s]_i$ et, quand il l'obtient, P_{4i} peut alors envoyer un nouveau $[NPQ]_i$; à la fin de la TEMP relative au premier des $(q+1)$ $[NPQ_s]_i$, P_{4i} automatiquement retransmet les $(q+1)$ $[NPQ_s]_i$, c'est-à-dire nous avons ce que nous appelons une retransmission globale ("go back" procedure (BURTON, 72 ; NERI, 77)) ; la retransmission globale permet de contrôler les situations dans P_{4i} consécutives à la non-obtention du premier des $[NPQ_s]_i$. Notons que l'état d'attente dans P_{4i} est identique que l'on soit avant ou après la retransmission globale.

La politique 2 a une conséquence immédiate : la nécessité pour P_{4j} de réordonner les $[NPQ_s]_i$ acceptés dans le désordre.

La troisième considération concerne, dans l'hypothèse du mode 2 et de la politique 2, les deux techniques possibles, pour P_{4i} , d'accuser réception des $[NPQ_s]_i$ acceptés : nous distinguons ce que nous appelons l'ACC collectif et le ACC individuel.

Avec la technique du ACC collectif, P_{4i} envoie un $[RP(ACC)]_j$ à un $[NPQ]_i$ accepté seulement si tous les $[NPQ_s]_i$ dont les numéros sont inférieurs à celui-ci ont été acceptés. Donc, avec cette technique, P_{4j} peut accuser réception d'un $[NPQ]_i$ accepté, de plusieurs façons : nous disons, relativement à un $[NPQ]_i$ qu'un $[RP(ACC)]_j$ est un ACC d'ordre 0 ou d'ordre 1 ou suivant qu'il accuse réception de ce $[NPQ]_i$ et aucun $[NPQ]_i$ avec un numéro supérieur ou qu'il accuse réception de ce $[NPQ]_i$ et du $[NPQ]_i$ avec le numéro immédiatement supérieur ou

Avec la technique du ACC individuel, P_{4i} accuse réception de chaque $[NPQ]_i$ accepté, à l'instant d'acceptation (comme dans le mode 1 et le mode 2 avec la politique 1).

La technique du ACC collectif a les conséquences immédiates suivantes :

- suivant que, P_{4i} obtiendra le premier des $(q+1)$ $[NPQ_s]_i$ ou ne l'obtiendra pas, P_{4i} enverra le $[RP(ACC)]_j$ relatif au premier de ces $(q+1)$ $[NPQ_s]_i$ ou n'enverra aucun $[RP]_j$ même s'il accepte des $[NPQ_s]_i$ parmi les q suivants,
- compte tenu de la manière de répondre de P_{4j} , on peut voir que le comportement de P_{4i} (cf phrase floue) doit être identique à celui indiqué dans le cas du mode 1 avec la politique 1 (remarquons : que P_{4i} ait ou n'ait pas accepté des $[NPQ_s]_i$ parmi les q $[NPQ_s]_i$ suivant le premier, P_{4i} ne le sait pas et donc la retransmission globale est le seul moyen de contrôler toutes les situations possibles en P_{4j}),
- l'état d'attente dans P_{4i} après la retransmission globale est différent de celui avant la retransmission globale (différence avec le cas du mode 1 et de la politique 1) c'est-à-dire maintenant P_{4i} attend n'importe quel $[RP(ACC)]_j$ relatif à n'importe quel des $(q+1)$ $[NPQ_s]_i$ retransmis (ou encore en parlant du premier des $(q+1)$ $[NPQ_s]_i$, P_{4i} attend n'importe quel ACC allant de l'ordre 0 à l'ordre q) : en effet, quand les $[NPQ_s]_i$ retransmis arrivent en P_{4j} , et, si le premier est maintenant accepté, P_{4j} peut, compte tenu des différentes possibilités de $[NPQ_s]_i$ acceptés précédemment à la retransmission globale, envoyer maintenant tout l'éventail possible des $[RP(ACC)]_j$; notons encore que, compte tenu de la sémantique du premier $[RP(ACC)]_j$ obtenu après la retransmission globale, P_{4i} peut alors envoyer plusieurs nouveaux $[NPQ_s]_i$ (en effet, $(q+1)$ $[NPQ_s]_i$ peuvent être mis en transit).

La technique du ACC individuel a les conséquences immédiates suivantes : même si P_{4i} n'obtient pas le premier des $(q+1)$ $[NPQ_s]_i$, P_{4i} enverra un $[RP(ACC)]_j$ relatif à chaque $[NPQ]_i$ acceptés parmi les q suivants ; donc P_{4i} doit prendre en compte n'importe quel $[RP(ACC)]_j$ relatif aux $(q+1)$ $[NPQ_s]_i$. On peut maintenant préciser le comportement de P_{4i} (cf phrase floue). Jusqu'à la fin de la TEMP associée au premier des $(q+1)$ $[NPQ_s]_i$, P_{4i} attend :

- a) le $[RP(ACC)]_j$ à ce premier des $(q+1)$ $[NPQ_s]_i$; quand il est obtenu, P_{4i} peut alors envoyer un nouveau $[NPQ]_i$,
- b) également, des $[RP(ACC)]_j$ relatifs aux q $[NPQ_s]_i$ suivants (la possibilité de cette obtention dépend évidemment de la durée de la TEMP considérée) ; ces $[RP(ACC)]_j$ sont enregistrés et les envois de nouveaux $[NPQ_s]_i$ consécutifs à ces obtentions de $[RP(ACC)]_j$ seront envisagés seulement après la fin de la TEMP considérée. A la fin de la TEMP associée au premier des $(q+1)$ $[NPQ_s]_i$, on retransmet uniquement ce $[NPQ]_i$ (retransmission sélective (METZNER, 77 ; EASTON, 79)).

La quatrième considération concerne dans l'hypothèse du mode 2 avec la politique 2 et la technique du ACC individuel, les deux possibles comportements de P_{4i} après la retransmission du premier des $(q+1)$ $[NPQ_s]_i$: comportement 1 ou comportement 2 suivant que P_{4i} n'envoie pas ou envoie un nouveau $[NPQ]_i$ pour chaque $[RP(ACC)]_j$ obtenu relativement aux q $[NPQ_s]_i$ suivants ; dans le cas du comportement 2 et durant la retransmission considérée, on peut avoir q nouveaux $[NPQ_s]_i$ envoyés (en effet $(q+1)$ $[NPQ_s]_i$ peuvent être en transit).

Cependant, concernant le comportement 2, on peut envisager le cas où P_{4i} retransmet toujours le premier $[NPQ]_i$ considéré et envoie de nombreux nouveaux $[NPQ_s]_i$. Ce comportement doit avoir des limites parce que, en particulier, l'ensemble des numéros utilisés pour la numérotation des $[NPQ_s]_i$ n'est pas infini. Nous appelons y le nombre maximum $[NPQ_s]_i$ qui peuvent être envoyés tant que le premier des $(q+1)$ $[NPQ_s]_i$ est toujours en transit ($y > q$). La valeur de y est une autre contrainte (après la valeur de q) d'une implémentation.

Ces quatre considérations nous amènent à définir les cinq stratégies suivantes : la stratégie I qui concerne le mode 1 ; la stratégie II qui est relative au mode 2 avec la politique 1 ; la stratégie III qui est relative au mode 2 avec la politique 1 et l'ACC collectif ; les stratégies IV et V qui concernent le mode 2, la politique 1, le ACC individuel et respectivement le comportement 1 et le comportement 2.

2.1.2. Généralités sur ces stratégies de retransmission

2.1.2.1. Les ressources nécessaires dans P_{4i} et P_{4j}

Le nombre de mémoires tampons et de temporisations dans P_{4i} est fixé par le nombre maximum de $[NPQ_s]_i$ en transit : 1 tampon et 1 temporisation dans la stratégie I; (q+1) tampons et (q+1) temporisations dans les stratégies II, III, IV et V.

Le nombre de mémoires tampons dans P_{4j} est fixé par le nombre maximum de $[NPQ_s]_i$ que P_{4j} peut accepter avant de pouvoir transmettre des $[PQ_s]_i$ à P_{5j} : 1 tampon dans les stratégies I et II; (q+1) tampons dans les stratégies III et IV; (Y+1) tampons dans la stratégie V.

2.1.2.2. Etats fondamentaux du transfert des $[NPQ_s]_i$

Ce sont les états d'attente, dans P_{4i} et P_{4j} , qui sont caractérisés par des "informations de contexte": la Fenêtre F_i dans P_{4i} et la Fenêtre F_j dans P_{4j} . Une fenêtre est un sous-ensemble de la séquence des entiers naturels : le plus petit nombre et le plus grand nombre sont respectivement appelés le coin gauche et le coin droit).

La fenêtre F_i comprend les numéros des $[NPQ_s]_i$ qui ont été envoyés et dont les $[RP_s(ACC)]_j$ sont attendus.

La fenêtre F_j comprend les numéros des $[NPQ_s]_i$ qui sont attendus et seront donc acceptés s'ils sont reçus.

Les fenêtres F_i et F_j traduisent donc l'état des ressources utilisées dans P_{4i} et P_{4j} pour le transfert des $[NPQ_s]_i$.

2.1.2.3. Progression du transfert des $[NPQ_s]_i$

Cette progression est caractérisée par l'évolution des états d'attente dans P_{4i} et P_{4j} , c'est-à-dire, plus précisément, par des changements dans les fenêtres F_i et F_j .

Ces changements dépendent à la fois des résultats du transfert des $[NPQ_s]_i$ mais également des relations du niveau 4 avec le niveau 5 (P_{4i} occupe ses ressources avec les $[PQ_s]_i$ venant de P_{5i} et P_{4j} libère ses ressources en transmettant des $[PQ_s]_i$ à P_{5j}).

Il est absolument essentiel que les changements dans les fenêtres F_i et F_j soient synchronisés.

Une règle fondamentale de cette synchronisation est : quand P_{4j} envoie un $[RP(ACC)]_j$, sa fenêtre F_j doit être positionnée de manière à ce qu'il puisse accepter le(s) $[NPQ_s]_i$ que P_{4i} peut lui envoyer quand il reçoit ce $[RP(ACC)]_j$.

2.1.2.4. Réponse envoyée par P_{4j} quand il rejette un $[NPQ]_i$ reçu

P_{4j} rejette un $[NPQ]_i$ reçu, chaque fois que le numéro de ce $[NPQ]_i$ est incompatible avec la fenêtre F_j .

Cette incompatibilité a une des causes suivantes :

- 1 - Ce $[NPQ]_i$ est un $[NPQ]_i$ précédemment accepté par P_{4j} mais le $[RP(ACC)]_j$, envoyé par P_{4j} , n'a pas été obtenu par P_{4i} parce que :
 - a) ou il a été perdu dans les niveaux inférieurs,
 - b) ou, son contenu, ayant été perturbé par la transmission dans les niveaux inférieurs, les erreurs consécutives n'ont pas été détectées par le test du bloc de redondance dans P_{3i} et affectent sa sémantique.
- 2 - Ce $[NPQ]_i$ est un $[NPQ]_i$ envoyé pour la première fois par P_{4i} mais :
 - a) ou, son contenu, ayant été perturbé par sa transmission dans les niveaux inférieurs, les erreurs consécutives n'ont pas été détectées par le test du bloc de redondance dans P_{3i} et affectent son numéro de manière à ce qu'il ne corresponde plus à un numéro attendu,
 - b) ou (seulement dans la stratégie II), ce $[NPQ]_i$ est n'importe quel des q $[NPQ_s]_i$ suivant le premier des (q+1) $[NPQ_s]_i$ lorsque ce dernier a été perturbé comme indiqué en 2.a),
 - c) ou (encore seulement dans la stratégie II) ce $[NPQ]_i$ est n'importe quel des q $[NPQ_s]_i$ suivant le premier des (q+1) $[NPQ_s]_i$ lorsque ce dernier a été perdu dans les niveaux inférieurs.

Quand P_{4j} a rejeté un $[NPQ]_i$ reçu, P_{4j} envoie comme $[RP]_j$:

- dans les stratégies I, II, IV et V, le $[RP(ACC)]_j$ relatif à ce $[NPQ]_i$,
- dans la stratégie III, le dernier $[RP(ACC)]_j$ envoyé.

Cette manière de répondre permet de contrôler les situations d'erreurs indiquées ci-dessus :

- si 1.a ou 1.b, P_{4i} obtiendra un $[RP(ACC)]_j$ attendu,
- si 2.a ou 2.b ou 2.c, P_{4i} obtiendra un $[RP(ACC)]_j$ non attendu, ne le prend donc pas en compte et donc retransmettra le $[NPQ]_i$ concerné à la fin de la TEMP qui lui est associée.

Notons que, si on considère qu'il n'y a pas d'erreurs non détectées par les tests des blocs de redondance dans le niveau 3, les situations 1.b, 2.a et 2.b n'existent pas (évidemment la validité de cette hypothèse dépend des performances de ces tests ; cette hypothèse est la plus souvent considérée par les personnes traitant des protocoles de communication mais, en toute rigueur, et en particulier, dans une étude de fiabilité, il faut considérer ces situations).

2.1.2.5. Codage des $[RP_s(ACC)]_j$

Les $[RP_s(ACC)]_j$ sont représentés avec les numéros des $[NPQ_s]_i$. Plus précisément, le numéro d'un $[NPQ]_i$ représente :

- le ACC à ce $[NPQ]_i$ dans les stratégies I, II, IV et V,
- le ACC d'ordre 0 à ce $[NPQ]_i$ dans la stratégie III.

2.1.2.6. Numérotation des $[NPQ]_i$

Elle doit être choisie de manière à ce que P_{4j} puisse détecter l'arrivée de $[NPQ]_i$ déjà acceptés. Ces nouvelles arrivées sont consécutives à la non-obtention, par P_{4i} , des $[RP_s(ACC)]_j$ envoyés. Afin de déterminer la numérotation, considérons la situation suivante :

- a) supposons tout d'abord que le numéro du premier $[NPQ]_i$ envoyé est le numéro 0 et que P_{4j} a accepté ce $[NPQ]_i$ mais également tous les $[NPQ]_i$ que P_{4i} peut envoyer sans recevoir le $[RP(ACC)]_{4j}$ relatif à ce $[NPQ]_i$ de numéro 0 ; nous représentons ci-dessous la fenêtre F_j caractéristique de cette situation ($[a, b \dots]$ signifie que des tampons sont disponibles pour recevoir les $[NPQ]_i$ de numéro $a, b \dots$) :

Stratégie I	$F_j = [1]$
Stratégie II	$F_j = [q+1]$
Stratégie III et IV	$F_j = [q+1, q+2, \dots, 2q+1]$
Stratégie V	$F_j = [y+1, y+2, \dots, y+q+1]$

- b) supposons de plus que le $[RP(ACC)]_j$ relatif au $[NPQ]_i$ de numéro 0 n'est toujours pas reçu par P_{4i} qui donc le retransmet toujours.

Ce $[NPQ]_i$ de numéro 0, en supposant que son numéro est reçu correctement par P_{4i} , ne doit pas être confondu avec les numéros des $[NPQ]_i$ que P_{4j} peut accepter. En conséquence, la numérotation doit être, au minimum :

stratégie I : modulo 2 ; stratégie II : modulo $(q+2)$; stratégies III et IV : modulo $(2q+2)$;
stratégie V : modulo $(y+q+2)$.

2.2 Classe 2

Compte tenu de la définition de cette classe, nous obtenons immédiatement, la stratégie I₁ (pour le mode 1) et les stratégies II₁, III₁, IV₁ et V₁ (pour le mode 2) dont les caractéristiques nouvelles par rapport respectivement aux stratégies I, II, III, IV et V sont :

- quand P_{4j} reçoit un $[SE]_i$, P_{4i} envoie un $[RP(RET)]_j$ qui comprend un numéro représentant le numéro du prochain $[NPQ]_i$ attendu en séquence,
- quand P_{4i} reçoit un $[RP(RET)]_j$: dans la stratégie I₁, P_{4i} entreprend la retransmission du $[NPQ]_i$ en attente d'accusé de réception ; dans les stratégies II₁, III₁, IV₁ et V₁, P_{4i} entreprend la retransmission globale dans les stratégies IV₁ et V₁, P_{4i} retransmet seulement le $[NPQ]_i$ dont le numéro est inclus dans $[RP(RET)]_j$ (ce numéro est le numéro du premier $[NPQ]_i$ en attente d'un accusé de réception),
- quand P_{4i} reçoit un $[SE]_i$, il agit de manière identique au cas b.

Dans ces stratégies, comme on a deux types de $[RP]_j$, le cardinal de l'ensemble des $[RP_s]_j$ est donc deux fois plus élevé que celui relatif aux stratégies définies au paragraphe 2.1.

Cependant, en ce qui concerne le mode 1, on peut définir deux nouvelles stratégies (stratégies I₂ et I₃) dont le cardinal de l'ensemble des $[RP_s]_j$ est identique à celui de la stratégie I :

- stratégie I₂ : elle utilise les mêmes $[RP_s]_j$ que la stratégie I₁ (le type $[RP(ACC)]_j$ de dimension 2). Ses caractéristiques sont :
 - quand P_{4j} reçoit un $[SE]_i$, il envoie un $[RP(ACC)]_j$ comme s'il rejetait un $[NPQ]_i$,
 - quand P_{4j} reçoit un $[RP(ACC)]_j$ non attendu, il renvoie le $[NPQ]_i$ en attente d'accusé de réception (dans la stratégie I₁, un $[RP(ACC)]_j$ non attendu a la sémantique d'une demande de retransmission),
 - quand P_{4i} reçoit un $[SE]_i$, il agit de manière identique au cas b.
- stratégie I₃ : c'est une stratégie qui peut être définie en supposant que la redondance utilisée dans le niveau 3 détecte toutes les erreurs affectant le contenu des $[NPQ]_i$ et des $[RP_s]_j$: dans cette hypothèse, les niveaux inférieurs au niveau 4 perdent seulement des informations mais ne laissent pas passer des informations erronées (en particulier, on peut dire que, lorsque P_{4i} reçoit un $[NPQ]_i$ dont le numéro est incompatible avec la fenêtre F_j , ce $[NPQ]_i$ est un $[NPQ]_i$ précédemment accepté mais dont le $[RP(ACC)]_j$ a été perdu).

La stratégie I₃ utilise deux types de $[RP]_j$, chaque type étant de dimension 1 ; on a un $[RP(ACC)]_j$ qui est utilisé pour accuser réception de n'importe quel $[NPQ]_i$; on a un $[RP(RET)]_j$ qui est utilisé pour demander la retransmission de n'importe quel $[NPQ]_i$. Les caractéristiques de fonctionnement sont :

- quand P_{4j} reçoit un $[SE]_i$, il envoie le $[RP(RET)]_j$,
- quand P_{4i} reçoit le $[RP(RET)]_j$, il renvoie donc le $[NPQ]_i$ en attente d'accusé de réception,
- quand P_{4i} reçoit un $[SE]_i$, il agit de manière identique au cas b.

2.3. Récapitulation

Le tableau récapitulatif de la figure 3 donne les différentes stratégies avec, d'une part, les éléments nécessaires à leur définition et, d'autre part, leurs propriétés.

CONCLUSION

La présentation effectuée dans ce papier fait apparaître, à notre avis, deux grands points d'intérêt.

Tout d'abord, l'utilisation d'un modèle hiérarchisé à trois niveaux (le niveau demandeur du service pour lequel les stratégies de retransmission sont implémentées ; le niveau qui implémente ces stratégies ; la machine de transmission utilisée par le niveau précédent afin de mettre en oeuvre ces stratégies) a permis d'effectuer, à notre connaissance pour la première fois, une présentation précise et exhaustive des stratégies de retransmission. Enfin, cette présentation des différentes stratégies est une étape essentielle avant de passer à deux étapes ultérieures qui sont l'implémentation de ces stratégies et la modélisation formelle de la vérification logique.

BIBLIOGRAPHIE

- BARTLETT, 1969 "A note on reliable full-duplex transmission over half duplex links", Commun. Ass. Comput. Mach., vol. 12, May 1969.
- BURTON, 1972 "Errors and error control", Proc. IEEE, vol. 60, n°11, November 1972.
- EASTON, 1979 "Design choices for selective-repeat retransmission protocols", Research Report, IBM Thomas J. Watson Research Center, Yorktown Heights, New York.
- GRAY, 1972 "Line control procedures", Proc. IEEE, vol. 60, n°11, November 1972.
- LUCKY, 1968 "Principles of data communication", New York, Mc Graw-Hill, Book Company Inc., 1968.
- LYNCH, 1968 "Reliable full duplex file transmission over half duplex telephone lines", Common ACM 11, 6 June 1968.
- MACCHI, 1979 "Téléinformatique : transport et traitement de l'information dans les réseaux et systèmes téléinformatiques", Dunod, 1979.
- METCALFE, 1973 "Packet communication", MAC TR-144, Massachusetts Institute of Technology, December 1973.
- METZNER, 1977 "A study of an efficient retransmission strategy for data links", NTC 77 Conf. Rec., pp. 3B : 1-1 to 3B : 1-5.
- NERI, 1977 "A reliable control protocol for high-speed packet transmission", IEEE Trans. on Communication, October 1977.

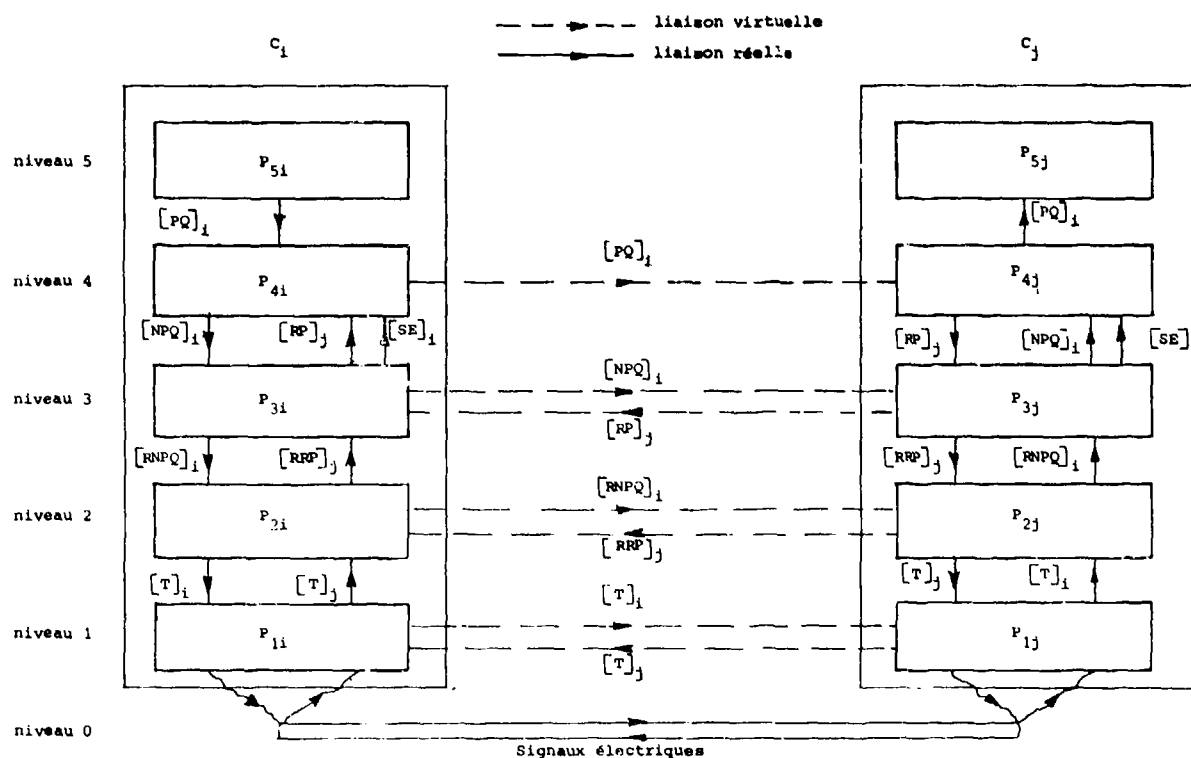


Figure 1 Modèle hiérarchisé du système considéré

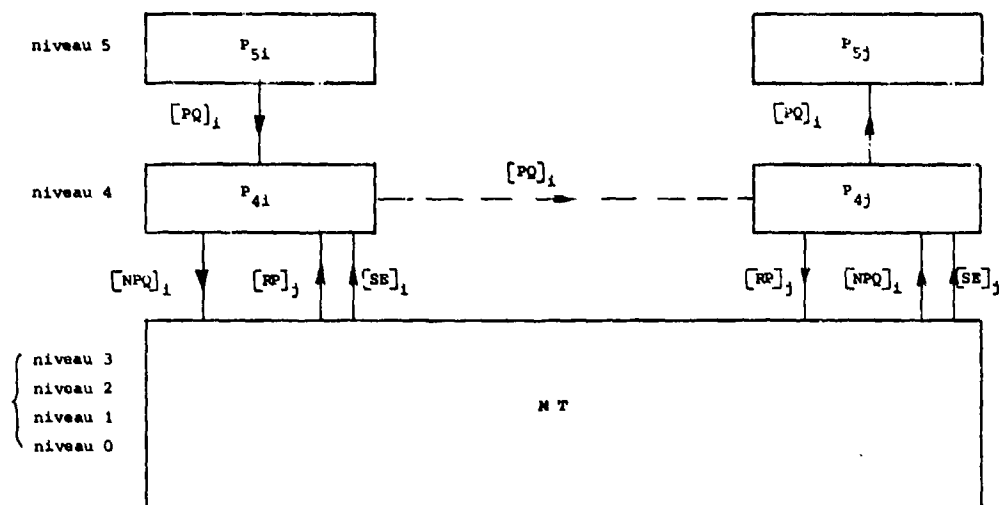


Figure 2 Modèle hiérarchisé considéré

Comment P_{4i} envoie les $[NPQ]_{i,j}$?	Comment P_{4j} accepte les $[NPQ]_{i,j}$?	Comment P_{4j} accuse réception des $[NPQ]_{i,j}$ acceptés ?	P _{4i} peut-il envoyer un nouveau $[NPQ]_{i,j}$ tant qu'il n'a pas reçu l'accusé de réception relatif au 1er $[NPQ]_{i,j}$?	Cause de la retransmission	Numéro de module min. max	Les réponses de P_{4j}		Stratégies
						les types	dimension de chaque type	
mode 1	en séquence	ACC individuel	non	TEMP	modulo 2	$[RP(ACC)]_j$	2	I
				TEMP, $[SE]_j, [SE]_i$		$[RP(ACC)]_j$		I_1
						$[RP(RET)]_j$		I_2
						$[RP(ACC)]_j$		I_3
mode 2	en séquence	ACC individuel	non	TEMP	modulo(q+2)	$[RP(ACC)]_j$	q+2	II
				TEMP, $[SE]_j, [SE]_i$		$[RP(ACC)]_j$		II_1
	dans n'importe quel ordre	ACC collectif	non	TEMP	modulo (2q+2)	$[RP(ACC)]_j$	2q+2	III
				TEMP, $[SE]_j, [SE]_i$		$[RP(ACC)]_j$		III_1
		ACC individuel	non	TEMP		$[RP(RET)]_j$		IV
				TEMP, $[SE]_j, [SE]_i$		$[RP(ACC)]_j$		IV_1
				TEMP		$[RP(RET)]_j$		V
				TEMP, $[SE]_j, [SE]_i$		$[RP(ACC)]_j$		V_1
		ACC individuel	oui	TEMP	modulo (y+q+2)	$[RP(ACC)]_j$	y+q+2	
				TEMP, $[SE]_j, [SE]_i$		$[RP(RET)]_j$		

Figure 3 Tableau récapitulatif

PRACTICAL ASPECTS WHICH APPLY TO MIL-STD-1553B DATA NETWORKS

by

Mr. I. Moir

Military Systems Engineer

Smiths Industries Aerospace & Defence Systems Company

Cheltenham Division, Bishops Cleeve, Cheltenham

Gloucestershire, GL52 4SF. England.

and

Mr. P.A. Duke

Senior Avionics Systems Engineer

British Aerospace, Brough

North Humberside, UK

SUMMARY

This paper discusses practical aspects which apply when attempting to design a complex avionics system based on a Data Bus Architecture. An example of such a system is the Stores Management and Weapon Aiming system, and this is discussed in detail.

1. INTRODUCTION

1.1 The Arrival of the Data Bus

The data bus offers many potential advantages over hardwired or dedicated data transmission systems in the design of Avionic Systems. Systems are interconnected by a single or redundant twisted pair of wires via standard interfaces, so reducing inter system wiring and the types and numbers of interfaces. The quantity of data transferred no longer has a direct influence on the inter system wiring and distributed computing becomes feasible. However in spite of the obvious advantages of a data bus system there are certain limitations which could be the source of much heartache to the system designer. Problems may result from transmission delays, digital sampling noise and the fundamental upper limit on data item-data rate product. Also, since interconnection between systems is via a common path, faults in the communication medium can have serious consequences, and therefore the use of redundancy and error correction techniques need to be employed.

1.2 Designing a New System

When starting to design a new avionic system based on the Data Bus principle a number of system configurations can be devised. The functional areas can be allocated to hardware units and the interface signals rationalised. However, the resulting system could present a high technical risk unless practical experience has been gained with the system, and the inevitable design problems identified.

In order to investigate the limitations of data bus systems, to gain practical experience of distributed computing centres interconnected by data buses (in advance of a new aircraft project) and to stimulate manufacture of bus compatible equipments, an Avionic Systems Rig facility has been established in the UK, at British Aerospace, Brough.

2. THE AVIONIC SYSTEMS RIG

2.1 History

The need for an Avionics Systems Rig became apparent during the 1970s, the intention being that the Rig would provide a tangible and cost effective means of risk reduction in the development of future aircraft avionic systems. The purpose is to demonstrate, in ground rig form, total system integration and system architectural design concepts, making use of the considerable technological progress which has been achieved in recent years, and in particular of the data bus.

2.2 Objectives and Aims

The next combat aircraft project was expected to appear during 1987 to 1990. Current uncertainties about timing and form of such an aircraft make a rig programme within the spirit of that originally conceived even more valid to exploit and practically apply developing technology.

The general objectives for the rig are:

- (a) To provide a focal point for the design, development and practical demonstration of a fully integrated total system using a multi bus architecture with sub system integration, asynchronous data transfers and total system executive control.
- (b) To support the design and development of systems for a fixed wing tactical combat aircraft to enter service towards the end of the present decade.
- (c) To provide a stimulus for the production of equipment compatible with DEF Stan. 00-18 (Part 2) (i.e. MIL-STD-1553B).

- (d) To investigate the specification, procurement and management procedures for a highly integrated system.
- (e) To effect an improvement in total system fault diagnosis.
- (f) To develop a capability for the control and management of software procurement and adherence to quality assurance standards.
- (g) To develop systems which will be properly matched to the pilot's requirements and capabilities.

2.3 Avionics Industry Participation

It was recognised from the outset that the UK avionics industry should be closely involved with the project. This has been achieved during the planning process through consultation with UK avionics companies. A working group comprising senior engineers from a number of these companies has been set up:

- (a) To ensure that the Rig reflects current developments in avionic systems technology.
- (b) To help in the communication of results and experience from the Rig program to the Avionics Industry.
- (c) To provide a forum for the discussion of Rig procurement difficulties.
- (d) To provide a forum for the discussion of standards applicable to the rig.

During the early stages of the programme the working group has assisted in establishing an overall system architecture, and producing outline specifications for its sub systems.

2.4 The Architecture of a Multi Bus Avionic System

The overall systems architecture was derived in the light of studies carried out by the UK aircraft and avionics industry over a number of years. One of the studies was to develop a systems architecture for an offensive support aircraft. This was carried out using a 'top down' approach to system design which led to functional grouping of equipments. These functional groupings were found to give advantages in the comprehension of system operation, the specification of system performance, and in equipment procurement and management. The functional groups derived were:

- (a) Aircraft Group
- (b) Pilot Group
- (c) Navigation Group
- (d) Mission Group

The Aircraft Group of sub systems comprise those sub systems which are primarily concerned with keeping the aircraft flying safely i.e. they are safety critical, and contain the Flight Control and General Aircraft systems.

The Pilot Group contains systems and functions which interface directly with the pilot, such as the cockpit controls and displays together with those such as the avionics bus controller which provide a total system control function.

The Navigation Group contains systems and functions which determine the position of the aircraft and where it is to go.

The Mission Group embraces all those functions that are concerned with attack, defence and stores management.

The systems within these functional groupings will communicate over the avionics bus as shown in Figure 1.

Whilst communication between groups takes place over the Avionics Bus it was found that for geographically distributed sub systems and units within groups additional data buses within the group were required. A particular instance which will be considered in more detail later in this paper is the Stores Management function within the attack area of the mission group.

The architectural configuration was driven mainly by availability and safety requirements. One requirement which was a strong driver was that wherever possible no single failure should cause a mission abort. Another was that single or combined failures should have a very low probability of hazarding the aircraft or friendly personnel on the ground. Groups of systems which have this safety requirement are shown at the bottom of Figure 1.

PRACTICAL ASPECTS WHICH APPLY TO MIL-STD-1553B DATA NETWORKS

by

Mr. I. Moir

Military Systems Engineer

Smiths Industries Aerospace & Defence Systems Company

Cheltenham Division, Bishops Cleeve, Cheltenham

Gloucestershire, GL52 4SF. England.

and

Mr. P.A. Duke

Senior Avionics Systems Engineer

British Aerospace, Brough

North Humberside, UK

SUMMARY

This paper discusses practical aspects which apply when attempting to design a complex avionics system based on a Data Bus Architecture. An example of such a system is the Stores Management and Weapon Aiming system, and this is discussed in detail.

1. INTRODUCTION

1.1 The Arrival of the Data Bus

The data bus offers many potential advantages over hardwired or dedicated data transmission systems in the design of Avionic Systems. Systems are interconnected by a single or redundant twisted pair of wires via standard interfaces, so reducing inter system wiring and the types and numbers of interfaces. The quantity of data transferred no longer has a direct influence on the inter system wiring and distributed computing becomes feasible. However in spite of the obvious advantages of a data bus system there are certain limitations which could be the source of much heartache to the system designer. Problems may result from transmission delays, digital sampling noise and the fundamental upper limit on data item-data rate product. Also, since interconnection between systems is via a common path, faults in the communication medium can have serious consequences, and therefore the use of redundancy and error correction techniques need to be employed.

1.2 Designing a New System

When starting to design a new avionic system based on the Data Bus principle a number of system configurations can be devised. The functional areas can be allocated to hardware units and the interface signals rationalised. However, the resulting system could present a high technical risk unless practical experience has been gained with the system, and the inevitable design problems identified.

In order to investigate the limitations of data bus systems, to gain practical experience of distributed computing centres interconnected by data buses (in advance of a new aircraft project) and to stimulate manufacture of bus compatible equipments, an Avionic Systems Rig facility has been established in the UK, at British Aerospace, Brough.

2. THE AVIONIC SYSTEMS RIG

2.1 History

The need for an Avionics Systems Rig became apparent during the 1970s, the intention being that the Rig would provide a tangible and cost effective means of risk reduction in the development of future aircraft avionic systems. The purpose is to demonstrate, in ground rig form, total system integration and system architectural design concepts, making use of the considerable technological progress which has been achieved in recent years, and in particular of the data bus.

2.2 Objectives and Aims

The next combat aircraft project was expected to appear during 1987 to 1990. Current uncertainties about timing and form of such an aircraft make a rig programme within the spirit of that originally conceived even more valid to exploit and practically apply developing technology.

The general objectives for the rig are:

- (a) To provide a focal point for the design, development and practical demonstration of a fully integrated total system using a multi bus architecture with sub system integration, asynchronous data transfers and total system executive control.
- (b) To support the design and development of systems for a fixed wing tactical combat aircraft to enter service towards the end of the present decade.
- (c) To provide a stimulus for the production of equipment compatible with DEF Stan. OO-18 (Part 2) (i.e. MIL-STD-1553B).

2.5 Reliability Requirements

The general reliability requirement for a mission critical avionic function is that no single failure within that function should prevent the completion of a mission. This requirement can be satisfied by simplex systems provided that an alternative (or reversionary) source of the data produced by that system is available. A second failure occurring within an avionic function can result in the failure of that function and cause the mission to be abandoned.

For a safety critical function there is an additional requirement that no single failure within that function shall result in a hazardous output.

A given function, may involve several systems (e.g. The Weapon Aiming Function may call for data from the Radar, Electro Optical Sensors, Navigation System, Air Data System and Pilot). The communication path between these systems needs to reflect the reliability requirements of a given function. Hence the avionics bus, which only transfers signals which are classed as being mission critical is itself mission critical and must be at least dual redundant. Within the weapon control and release area however two functions with different reliability requirements come together to produce the successful release of a weapon. They are the weapon aiming function, which is mission critical, and the weapon release function, which is safety critical.

A discussion concerning the design options available for a future weapon system forms the main contents of this paper. It has been written jointly by BAe Brough and Smiths Industries, who have for some time been working together on the design and use of MIL-STD-1553B data bus transmission systems.

3. THE WEAPON SYSTEM

3.1 Introduction

On the majority of aircraft, weapons are carried externally, on wing and fuselage pylons. There may be eleven or more weapon stations and each may carry more than a single weapon. When considering a new weapon system a major design consideration is how to communicate between the sensors and weapon system processors and the weapons. Many alternative configurations are possible and it has been argued that it would be more cost effective to employ a current digital data transmission system which has been proved, than to develop a new one. However, this paper is exclusively concerned with data bus issues and therefore existing data transmission systems have been excluded.

The signals required by a weapon can be divided in three general classes: Aiming, Arming and Release.

3.2 Weapon Aiming

The weapon aiming system will obtain information from a number of sources to provide:

- (a) Attack geometry to the flight control system.
- (b) Display information to the cockpit.
- (c) Release cues to the weapon release system.
- (d) Guidance signals to the weapon heads.

Many weapon types need to be told where their targets are. Additionally they may require such data as aircraft altitude, velocity, target velocity etc. This data is dynamic, is generated by the combined operation of several systems (e.g. Navigation Radar and EO systems and the pilot). These signals demand high data rate and are mission critical.

As the guidance signals result from the combined operation of a number of systems, an early consideration involves the distribution of processing between the various systems which contribute to the weapon aiming function. The centralised and distributed weapon aiming system architectures are shown schematically in Figures 2, 3 and 4.

A centralised aiming system has the advantage of having all the data required for aiming calculations available within a single unit. However, this data must be transferred from the sensor and using a data bus as shown in Figure 3 can introduce delays and digital sampling noise onto the highly dynamic data. The loss of accuracy and noise could be removed by using dedicated links from the sensors but this solution should be discouraged as the proliferation of dedicated links destroys the advantages of using data buses.

If we consider air-to-air and air-to-ground weapon aiming separately then we find that various systems are already performing many of the calculations required for weapon aiming. Thus with minimal additional processing the radar, for example, could perform the majority of the air-to-air Weapon Aiming computation and the Navigation system could perform the majority of the air-to-ground Weapon Aiming computation. On balance the distributed processing option shown in Figure 4 is preferred.

In each case the guidance signals must be transferred to all weapons. To add 11 or more extra remote terminals onto the Avionics bus would exceed the limit for a single bus and for this reason above we are forced to add an extra bus to transfer the aiming data.

3.3 Weapon Arming

Arming signals include bomb and missile fuze selection, missile priming functions such as thermal battery initiate, and the switch on of aircraft supplied electrical power. These are generally discrete signals which place the weapon in an active state, initiate electro-explosive devices or control the action of electro explosive devices. They require a low data rate but are both mission and safety critical.

3.4 Release Signals

In general there are three different types of release signal, which are classified according to their form at the aircraft to weapon interface.

- Type 1 Release Signal

In order to release a bomb it is necessary to apply a high current discrete to the Ejector Release Unit mounted in the aircraft pylon or multiple carrier unit. No signal passes from the aircraft to the external store.

- Type 2 Release Signal

In order to release a missile it is necessary to apply a high current discrete signal to the rocket motor igniter. This discrete must be passed from the aircraft to the missile.

- Type 3 Release Signal

For many small weapons carried in large numbers on a special carrier with release under the control of a unit mounted within the weapon carrier. The release signal is a low current discrete or digital data word which is passed from the aircraft to the carrier.

Whichever type is required the release signal is of very low data rate, mission critical and safety critical and will have to be applied at a precise time.

With the addition of the weapon aiming bus we must now consider the weapon release and arming functions and the options available for their implementation.

4. BUS REALISATION OPTIONS AND THE EFFECT UPON SUBSYSTEM DESIGN

4.1 System Requirements

(a) Safety

- (i) 'No single failure shall result in a safety critical signal being generated.' This generally means that signals such as release of stores, etc. must operate effectively in a duplex mode i.e. two independent signals must be present before release or other safety critical function can occur.
- (ii) 'The probability of dormant or multiple failure modes resulting in a safety critical output shall be extremely small.' (The figure of 1 in 10^{-7} per flight hour is often quoted.)

(b) Availability

Existing Stores Management Systems typically specify that the probability of a failure to operate should be not greater than a certain figure, and in this context a figure of 1 in 10^{-4} per flight hour is often quoted.

Bearing in mind that current systems are usually designed to provide alternative (reversionary) methods for releasing weapons and that it should be the aim as far as possible to minimise the need for such reversion in future designs (which utilise data buses), the requirement for availability will be somewhat more stringent than the above figure suggests. However, the main avionics system will have reversionary modes enabling weapon release points to be computed using various sensors subsequent to sub system failures. It is essential therefore that the Stores Management System should be capable of exploiting this capability. A requirement for further methods of weapon release which exclude communication via the Avionics Bus needs to be questioned critically since this will involve the introduction of dedicated hardware with its appropriate cost and weight penalties.

(c) Survivability

The term survivability means mission survivability in an environment where there is a high probability of battle damage being experienced. Given that the effective release of the stores and missiles would be inadequate without the support of some of the aircraft sensors and other sub systems, then crude reversionary mechanisms

would be unlikely to contribute significantly to mission success. The potential of a dual redundant or similar data bus for providing both survivability and availability therefore needs to be fully exploited to match the avionic system capability.

4.2 Redundancy Options

The options available for dual operation may be summarised as follows:

(a) Duplex Redundancy

Both elements of a duplex redundant system have to be fault free to obtain full system performance. This may be likened to Figure 5A where two switches are connected in series.

(b) Dual Redundancy

In a dual redundant system either of the two elements can perform the same specified system function. This form of redundancy may be Active (e.g. cyclic redundancy, in which the elements are switched from one to the other and back again) or Passive (e.g. stand by redundancy, in which one element is active until a failure occurs in which case the alternative element is used). Dual redundancy is akin to the parallel switches shown in Figure 5B.

The safety requirements of a weapon release system dictate the operation of the elements in a duplex manner. The availability and survivability requirements demand some form of dual redundancy. Therefore the overall requirement will necessitate both duplex and dual redundant features such as shown in Figure 5C.

4.3 Bus Realisation

Duplex operation in a time division multiplexed digital transmission system (i.e. 1553B) may be implemented by time separation of identical messages down a data bus instead of duplicating hardware. By suitable coding of the independently generated duplex signals into separate words and transmitted over a single data highway to a RT and subsequently decoded back into true duplex signals, then it can be shown that for a bit error rate of 10^{-12} , the occurrence rate of valid duplex word error is of the order of 10^{-18} per hour. Indeed even if noise burst occur at higher levels than this, then providing reasonable temporal separation of the duplex words is made, the occurrence of error is still highly improbable. It seems reasonable then that techniques such as this may be implemented to protect against inadvertent release due to noise.

If the duplex words representing the release signal are dissimilar in bit structure, then it is reasonable to suppose that a single failure mode will not be able to generate spuriously both words. To prove this, however will require a rigorous failure mode analysis on the simplex part of the system, and this can only be done on a reasonably detailed design. Therefore whilst preliminary investigation indicates that safety critical signals may be transmitted satisfactorily via a dual redundant bus system, acceptance of the system will lean heavily on a detailed failure mode analysis followed by supporting experimental evidence.

A high weapon delivery availability (failure to operate not greater than 10^{-4} per hour) consistent with a high level of signal integration into the bus system, make it necessary to analyse the proposed system on the basis of random failure rates. The presence of dual computing and a dual redundant bus system should enable this requirement to be met in the configuration shown in Figure 6. The attractions of this configuration are as follows:

- (a) Duplex operation may be accommodated using one set of hardware by transmitting independently generated signals down a single highway and combining into true duplex signals in the pylon interface unit.
- (b) The existence of dual redundancy within the 1553B buses and RT hardware permits availability and survivability requirements to be met.
- (c) The architecture includes separate signal paths and processing areas for both weapon aiming and weapon release functions which may aid certification. However, the option is available to combine both aiming and release functions on the same bus should experimentation suggest this to be a sensible alternative.

5. INTERFACING WITH EXTERNAL STORES

As a result of the above discussion we now assume separate Weapon Aiming and Weapon Release Buses (remembering that current weapon release DDTS systems are excluded from this paper). There remains one link in the chain from the sensors and pilot to the weapons. That is the interface between the weapon buses and the weapons. Here again several design options, involving the partitioning of processing elements are available.

The use of data buses offers the possibility of obtaining a standard interface connector such as that proposed in MIL-STD-1760. This is highly desirable as it will improve interoperability. However, as we shall see, this is not easy to achieve.

5.1 External Store Types

A wide range of external stores are now carried by aircraft and the pace of new weapon development and the complexity of those weapons are continually increasing. Store types which a new aircraft can be expected to carry include:

- conventional 'iron' bombs
- cluster or dispenser weapons
- short range and medium range IR and Radar guided missiles
- Fuel Tanks
- Electro Optic sensor and designator pods
- Electronic warfare pods

The signals required by these stores cover many different types, but typically:

- 28 V low current and high current discretes
- Analogue signals
- Digital data highways of various types
- 28 V and 115 V power supplies
- Video
- R.F.

They are generated at each store station within Pylon Interface Units from signals transmitted via the Weapon Release and Weapon Aiming buses.

The aim of any new system should be to provide a standard interface which can accept current and future store types with the minimum of hardware modification.

To provide an example of the interface design process and the options available we will look at the installation of an AIM-9 missile. This is a typical guided weapon and can be carried singly or on multiple carriers.

5.2 Pylon/Launcher Configurations

Figure 7 shows a possible configuration for a simple weapon such as the AIM-9 Sidewinder. Weapon aiming (head slaving) data is fed into the pylon processor via a dual redundant weapon aiming bus. Firing signals to the ERU are fed from both release processors in order that emergency jettison may be achieved in the event of either processor failure. Discrete and head aiming signals are routed to the store in a conventional manner and existing launchers could be used. The advantages and disadvantages of this scheme are:

Advantages

- Uses existing launcher hardware, therefore no cost increment

Disadvantages

- Non standard interfaces Launcher/store
- Power lines and head aiming routed via pylon processor
- Expensive in RT hardware (2 RTs for headaiming signals might not be justifiable)

Figure 8 shows a modified arrangement where the aiming signals are routed directly to the launcher without interfacing with a processor in the pylon. This arrangement would require a Remote Terminal and processor in launcher to receive and generate head aim signals. This would require a simplex RT for simplex generation, or a dual-redundant RT if dual-redundant weapon aiming generation were preferred or justified. The advantages and disadvantages of this layout are:

Advantages

- Discrete and head aiming signals not routed via pylon processors, therefore cheaper aircraft equipment
- Common pylon to launcher interface possible
- Reduction in hardware possible (1 RT for head aiming may be justifiable)

Disadvantages

- Requires new or modified launcher, with resulting cost increase
- New or modified launcher not common with old launcher at pylon to launcher interface and therefore not compatible with existing aircraft

Figure 9 shows a multistore launcher capable of carrying a number of smart stores, each of which is interfaced to the launcher by means of a standard stores interface. As for the previous option the head aiming and discrete lines are consolidated in the launchers before being routed to the store(s). The advantages and disadvantages of this configuration are:

Advantages

- Could be standard with new AIM-9 launcher interface
- Standard launcher/store interface
- Dual redundant 1553B bus routed to multiple 'smart' stores

Disadvantages

- Expensive in terms of hardware - bus extender and standard interfaces
- Several missiles now share one electronics unit which becomes a common failure point

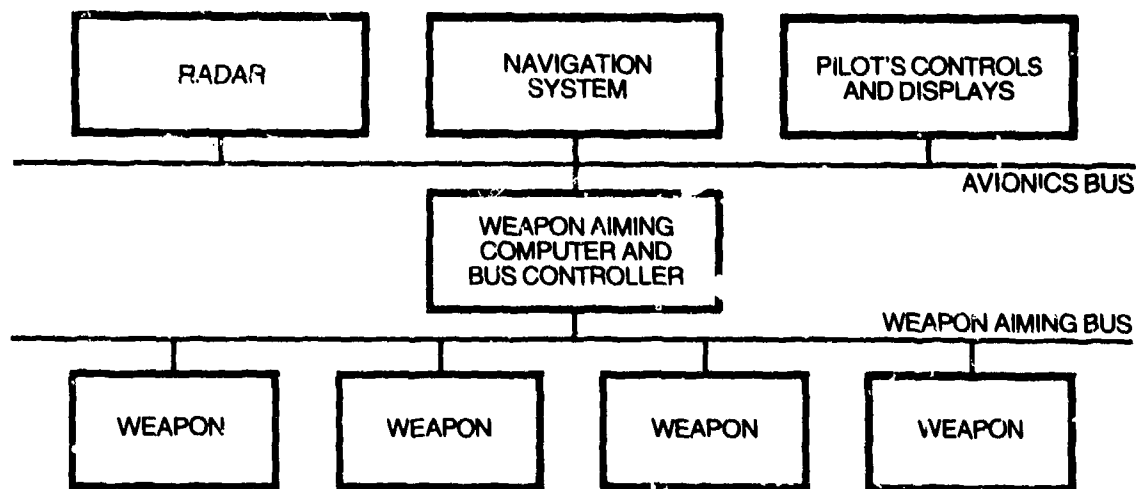
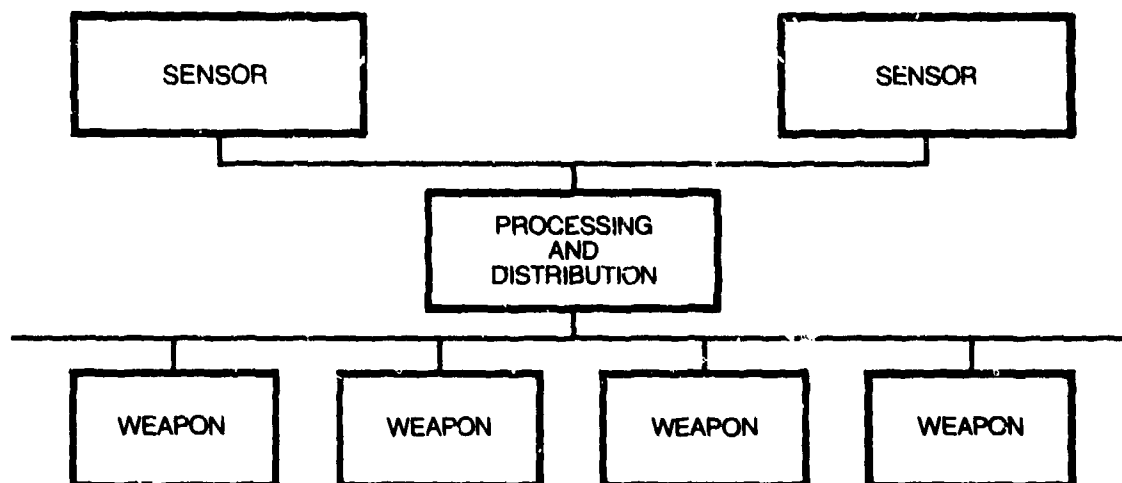
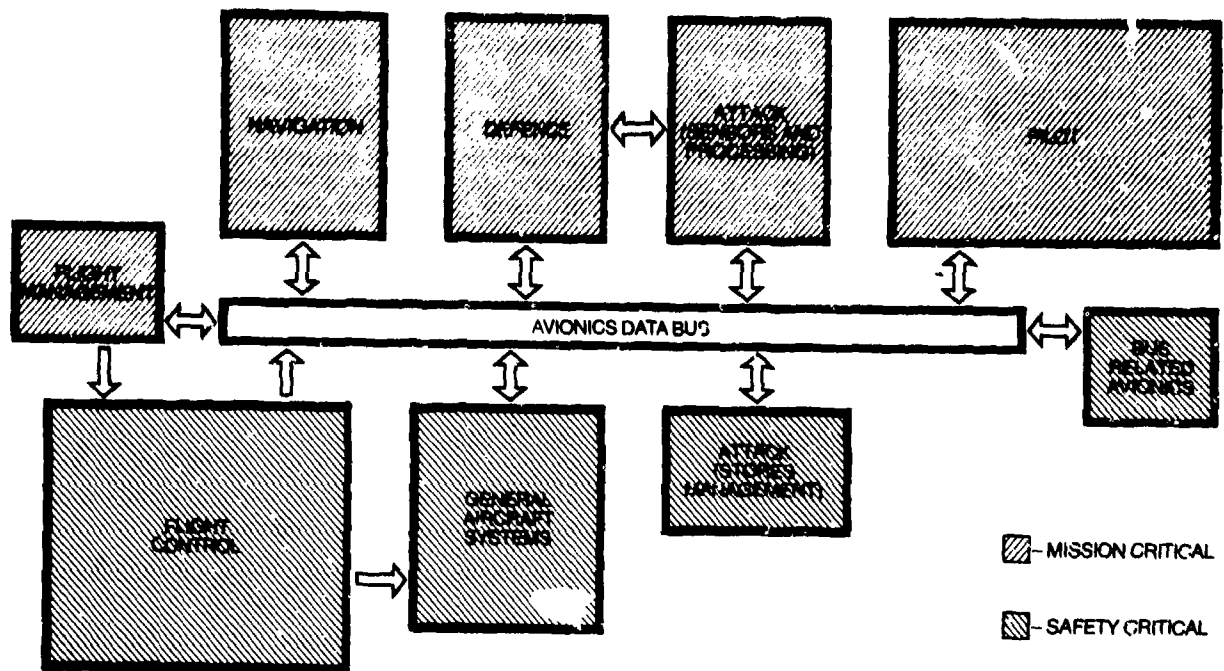
It may be seen that this configuration could easily be made compatible with the 'new' AIM-9 launcher configuration just described. Hence standard pylons could be interfaced with interchangeable launchers permitting a wide mix of weapons options to be carried. Furthermore, the use of standard stores interfaces at the launcher/store interface means that a mix of smart or dumb weapons of differing National ordnance could be carried. This feature would greatly enhance the effectiveness of NATO airborne tactical forces in any future conflict. The penalty paid for this interoperability is however the introduction of new and more complex launcher hardware including the incorporation of micro-processor hardware.

The technicalities associated with the 'bus extender' facility shown in the multiple MIL-STD-1760 launcher are presently being examined by Smiths Industries, in order to fully identify and quantify the trade-offs which are involved.

It may be seen that each of the three simplified options has advantages and disadvantages. Depending upon the need for carriage of smart weapons, and the need for standardization of the stores interface, trade offs exists in the areas of hardware complexity, weight, reliability, integrity and cost (including cost of ownership). The user will therefore need to list the relative priorities in these areas and quantify the trade-offs in order to assume maximum cost-effectiveness of the overall weapons system.

CONCLUSION

The paper has served to describe the work being undertaken in the UK industry on the interfacing of MIL-STD-1553B buses. The development of an integrated systems rig has been outlined and the aims of the rig identified. The use of data buses for weapon aiming and weapon release purposes have been described in detail. The trade-offs which exist when the requirements of a standard launcher/stores interface have also been taken into consideration.



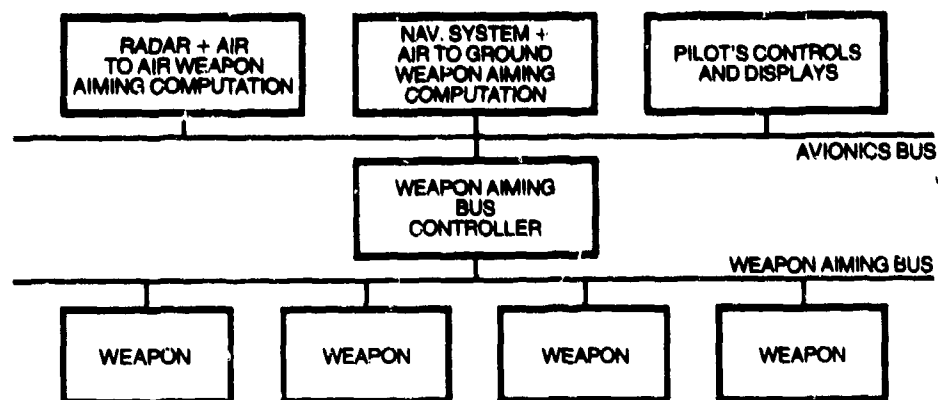


FIG. 4 DISTRIBUTED WEAPON AIMING SYSTEM

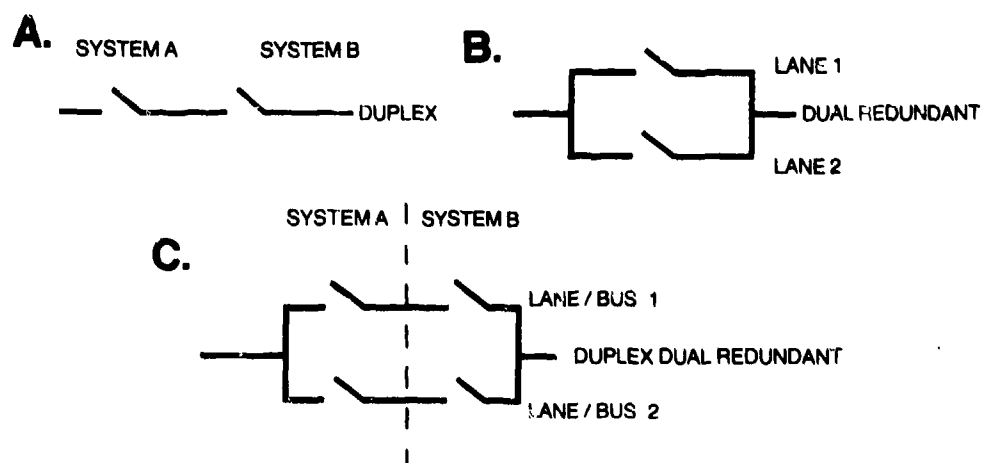


FIG. 5 DUAL SYSTEM OPERATION OPTIONS - SIMPLIFIED SCHEMATIC

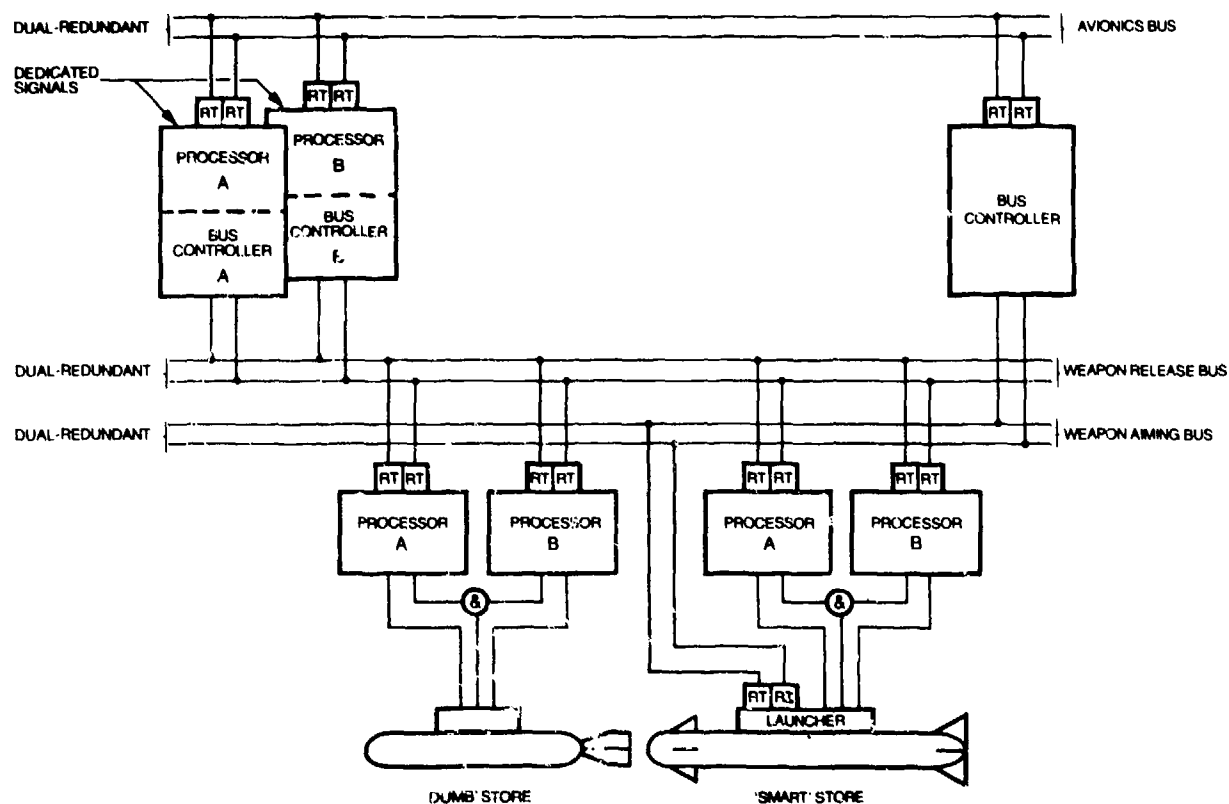


FIG. 6 GENERALISED WEAPON SYSTEM CONFIGURATION

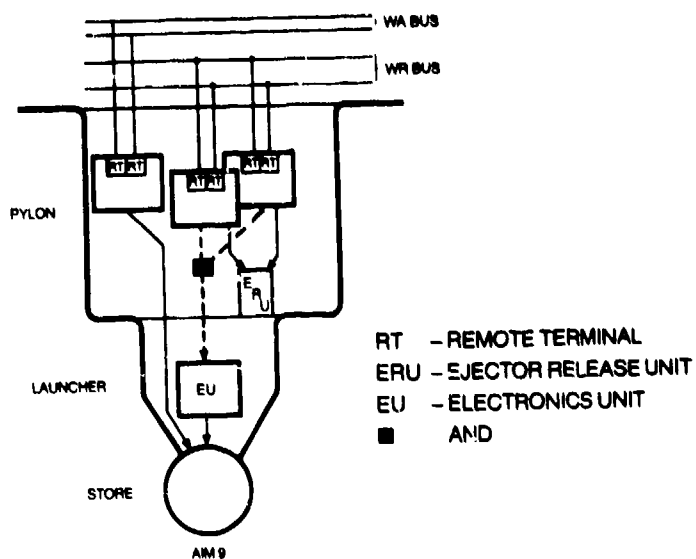


FIG. 7 PYLON INTERFACE OPTION 1

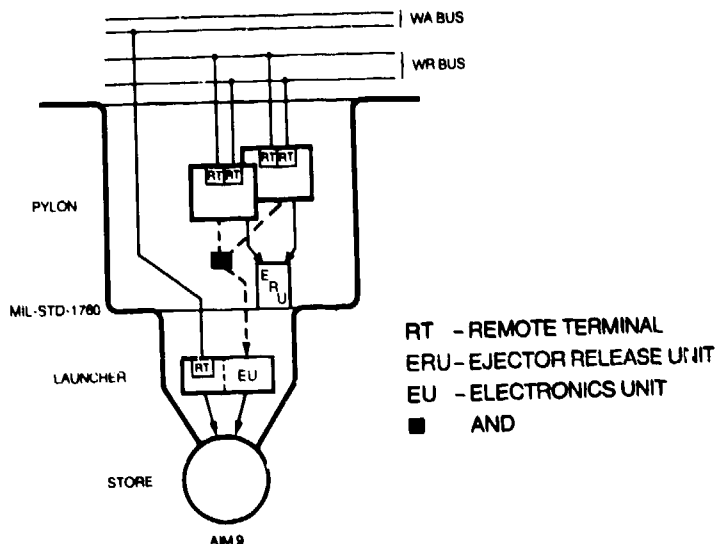


FIG. 8 PYLON INTERFACE OPTION 2

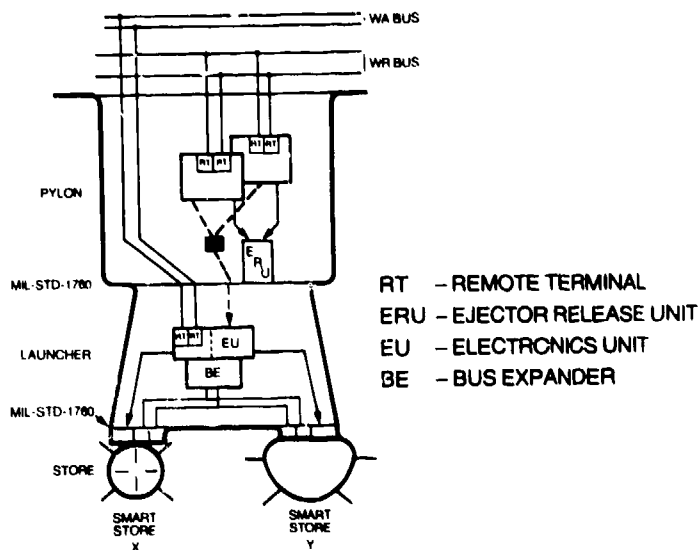


FIG. 9 PYLON INTERFACE OPTION 3

THE TRAFFIC FLOW IN A DISTRIBUTED REALTIME COMPUTING SYSTEM (RDC-SYSTEM) WITH A FIBEROPTIC RINGBUS SYSTEM

Dirk Heger and Reinhard Bähre

Fraunhofer Institut für
Informations- und Datenverarbeitung (IITB)
Sebastian-Kneipp-Str. 12/14
D-7500 Karlsruhe 1, Germany

ABSTRACT

The new generation of automatic systems is essentially characterized by distributed multi-computersystems. The architecture is based on distributed microcomputer stations linked together by a bus system. These systems give much more design alternatives than conventional single or multicomputer systems; the danger of obtaining bottle necks of system performance is considerably greater than it was by using functional modules operating independently and simultaneously. Therefore, mathematical modelling of bus-linked multicomputer systems and the experimental evaluation of these models in online operation by means of measurements is of increasing importance.

In this paper the RDC-system, a realtime computing system developed by the IITB and the traffic flow on its fiberoptic ringbus system are presented.

1. INTRODUCTION

The new generation of automatic systems is essentially characterized by distributed multi-computersystems (Fig. 1 and /1/). In systems like this use is made of a hierarchical decomposition of all tasks of the automating system. The decomposed tasks are distributed among a hierarchy of autonomous subsystems communicating with each other. These subsystems are realized by means of microcomputer stations today. Referring to an automatic fire control system we have for instance three functional levels performing the following tasks:

- Weapon control level controlling directly the weapons and the equipment in the field,
- fire control level coordinating several weapon control systems,
- command level for the orientation of all weapons in the battle field.

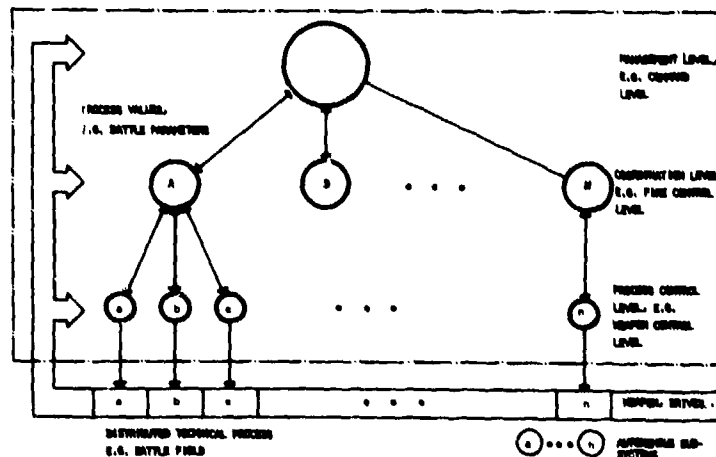


Fig. 1: Automatic control system,
(e.g. fire control system)

Fig. 2 shows a typical architecture of such automating systems. There we see the whole system which is composed by several subsystems called equipments. The equipments are coupled with each other by means of bus couplers to a bit serial system bus. This system bus for instance makes the connection to a central master control panel. The equipments again comprise subsystems which are called devices. These devices are linked together by the equipment bus, in general a parallel bus system. The devices in turn are built up by printed circuit boards which are connected by a parallel device bus. In the next step down we find components on the boards tied together again by a parallel bus system called board bus. And finally, we see clusters of I/O-devices which are connected with each other and the boards by an I/O-bus in turn.

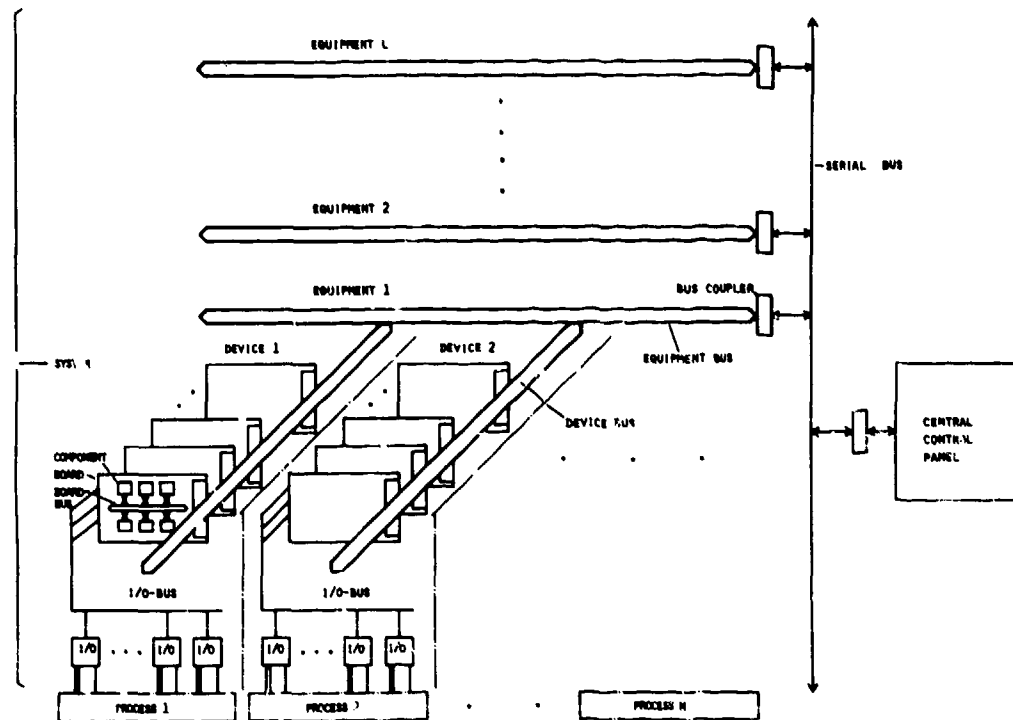


Fig. 2: Bus hierarchy in a distributed automating system

Let's summarize:

The hierarchy of the automatic control system is mapped onto a hierarchy of hardware subsystems in a dual manner [2]. Clusters of subsystems are pooled together by means of bus systems and form the subsystems belonging to the adjacent level of the hierarchy higher up. In any case, the functions of the subsystems should be performed autonomously in the main and the need of communication between these autonomous subsystems should be minimized in order not to get bottle necks of system performance and/or system availability. Typical lengths of the shown busses are for the device bus several 0.1 meters, for the equipment bus about 10 meters and for the serial system bus up to several kilometers. Distributed systems as described above are particularly important in cases in which the technical process is locally distributed. In these cases it is possible to limit the effects of failures locally and to replace the crashed functions by other parts of the system. This principle of error recovery or system reconfiguration with graceful degradation uses the principle of dynamic and functional redundancy.

2. THE DISTRIBUTED, FAULTTOLERANT RDC-SYSTEM

The Fraunhofer Institute for Information and Data Processing has developed the "Really Distributed Control Computer System", it is called RDC-system /3/,/4/. In this system there exist a lot of distributed microcomputer stations communicating with each other via a fiberoptic ringbus system. Up till now five RDC-systems are put into operation in different industrial applications, one of them for the closed loop control of 28 pit furnaces. The latter system has been working at the iron works at THYSEN AG since June 1979. This application will be described in the following, abbreviations used in the text and the pictures are listed in table 1.

LpP	Communication processor
LASP	Working storage of the LpP
SEA	Transmitter/receiver adaption
LSEM	Light transmitter/receiver module
PuP	Process control processor
PASP	Working storage of the PuP
BS/SR	Bus switch unit/fault diagnosis
NTST	Supply control
AA	Digital-to-analog converter
AE	Analog-to-digital converter
BA	Binary output
BE	Binary input
SBF/SBFT	Local control panel
E/A	I/O-devices
M	Sensors
St	Actuators
DSG	Alphanumeric terminal
MB	Magnetic tape recorder
MSP	Mass storage in the master control room
EAF	Color screen I/O panel
ZE	Computer for the master control operation with the color screen I/O panel
PR 310	Microcomputer SIEMENS 310

Table 1: Abbreviations

2.1 HARDWARE STRUCTURE

The hardware structure of the RDC-system is shown in Fig. 3. Here we see the distributed microcomputer stations with the microcomputers (PuP) performing the control tasks, the I/O-devices (E/A) and the measuring and actuating elements at the technical process (M,ST). Each station is connected by means of a special communication processor (LpP) to a bit serial fiberoptic ringbus system. In the middle of each station there is a special device (BSU) performing the on-line error detection within the station and at its interfaces, initializing the status reporting messages conveyed by the LpP to the whole system and being able to isolate the faulty parts. By this a systemwide distributed on-line fault-diagnosis is performed and the reconfiguration procedure is started. In the case of the pit furnace application the neighbour station replaces the functions of the faulty station and controls both technical processes simultaneously, but with degraded performance for each of them. After repair system regeneration automatically takes place as well. Moreover, we see at the top of Fig. 3 a double computer system. One of these computers performs all functions of displaying the statuses of the technical process and the automating system as well, the interactive operation by the operators and all documentation and data gathering tasks being essential for the next levels in the functional hierarchy. It is called EAF-system. The other computer fulfills in normal cases the task of dynamic loading

and starting of the RDC-stations in dependency of the system status. Moreover, it is the program production system for "MULTICOMPUTER-PEARL"-programs /5/. This "MULTICOMPUTER-PEARL" was developed by the Fraunhofer Institute and the German software house MBP/Werum.

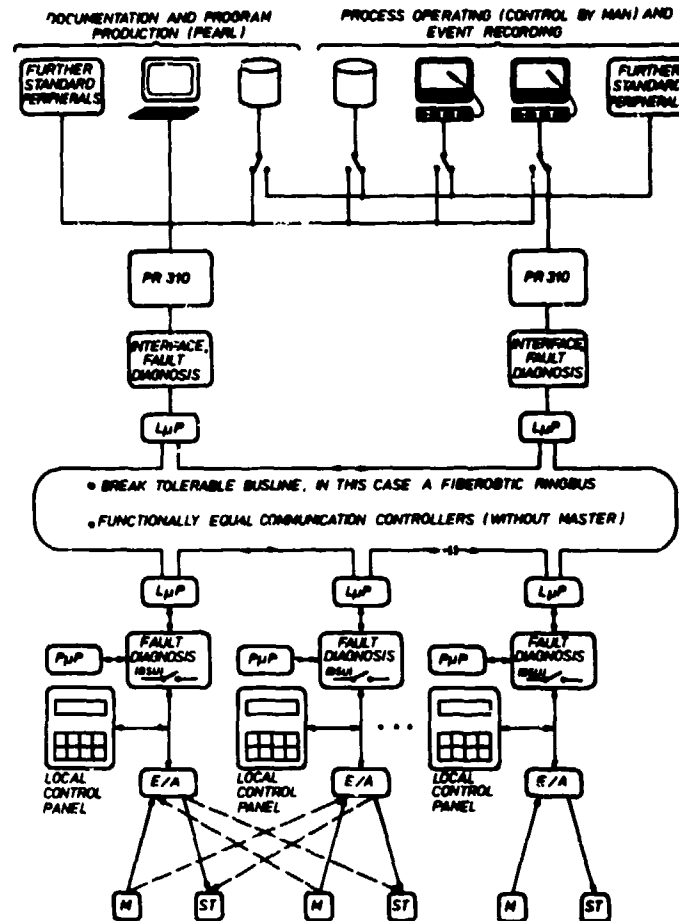


Fig. 3: The distributed, fault tolerant RDC-system (hardware structure)

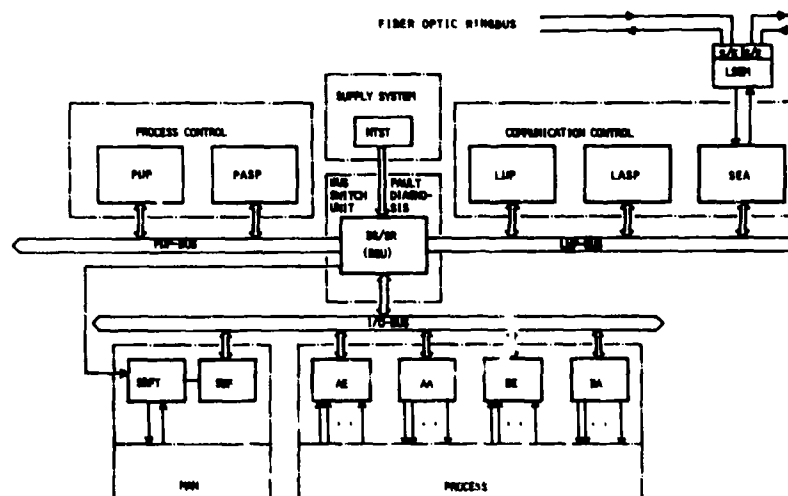


Fig. 4: Structure of a RDC-station

If one of these two computers fails the latter tasks are shut down and the remaining facilities are used to maintain the more important tasks of displaying the technical process and the interaction of the operating personnel. Besides the facility of central operation there exists the possibility of distributed input and output actions by an operator at the RDC-stations in the field. For this reason, each RDC-station has a local control panel. Fig. 4 shows the detailed internal hardware structure of a RDC-station.

2.2 PROGRAMS

Fig. 5 shows the distribution of the program system in the RDC-system. Each station contains identical transport systems, status reporting systems and line reconfiguration systems. All these programs are microprogrammed and performed by the LuP.

The RDC-stations contain identical network operating systems, local PEARL operating systems and run time systems in the levels higher up. These programs reside in the PuP and are called DISPOS (Distributed PEARL Operating System). And finally, all application programs are residently loaded for both cases of normal and reconfigured operation. Both computers of the central master control room contain the transport system, the status reporting system and the line reconfiguration system running on their LuPs. Higher up we see the network operating system with a local observer and the manufacturer supplied operating system ORG 310. And finally, there are the application programs of the EAF system and the program production and loading system.

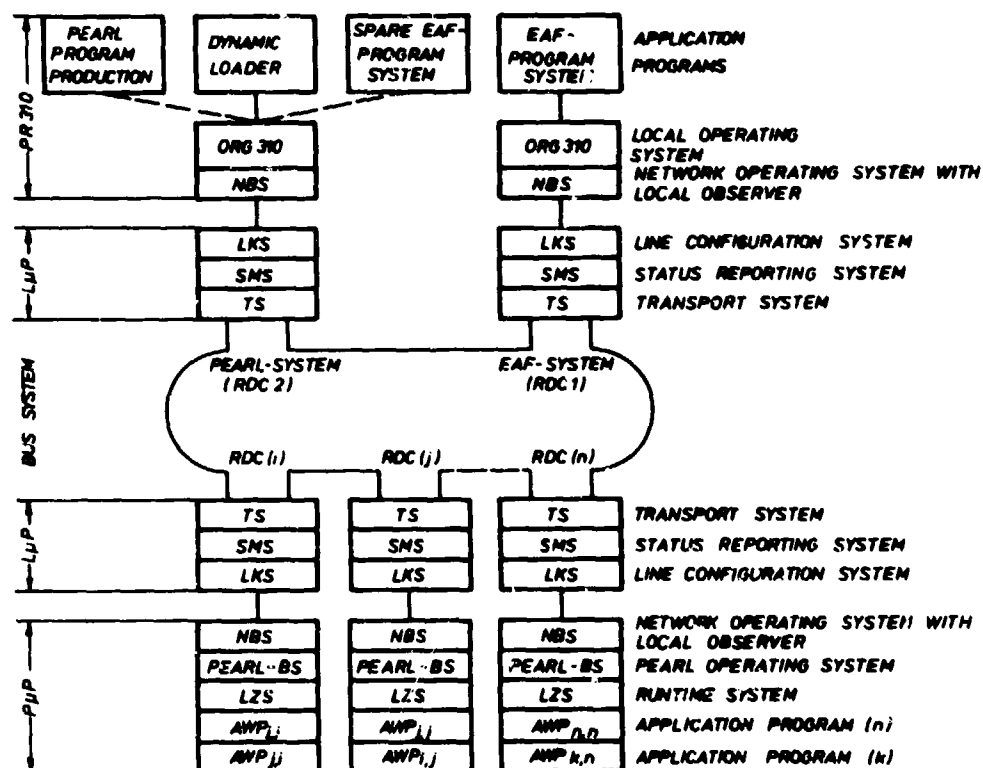


Fig. 5: The distributed program system of RDC

2.3 DATA BASE

Fig. 6 shows the distribution of the data base of the RDC-system. Each station has a list of system statuses, a list of actual process values, a list of control parameters and a list of internal process data. Moreover, each station has access to the actual values coming from the technical process. Copies for initialization are located on the bulk memories

of the master control computers. There exists an additional copy for reconstruction of the data base in case of a complete crash of the central computer system for a time.

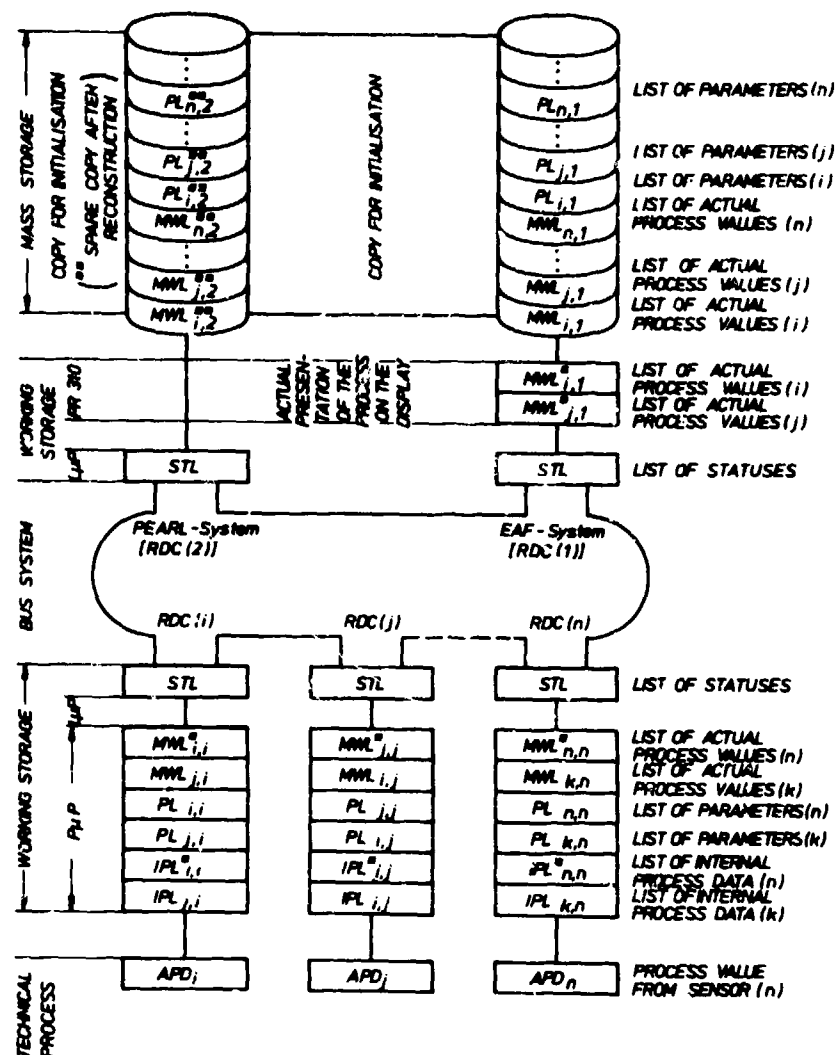


Fig. 6: The distributed data base of RDC

2.4 SUMMARY OF THE MOST IMPORTANT FEATURES

The most important features of the RDC-system are

- dynamic redundancy for all system levels and modules;
- fault tolerance, distributed fault diagnosis, also transmitted to and displayed on the central colour screen system;
- ODD-type bussystem:
 - Decentralized channel assignment
 - Decentralized message transmission
 - Decentralized message absorption
- transmission medium: optical fibers
- transfer rate: 10^6 bit per sec;
- distributed, hierarchical data base;
- MULTICOMPUTER-PEARL,
 - dynamic loader for automatic on-line loading;
- central control panel with flow chart representation with realtime update, curves and so on,

rolling map, light-pen interaction with operator guidance,
automatic selection of display information at changes of the process state.

3. CLASSIFICATION OF TECHNICAL COMMUNICATION SYSTEMS WITH RESPECT OF TRANSPORT AND CONTROL PRINCIPLES

Essential part of the RDC-system is its communication system. In the following a classification scheme for transport and control principles in technical communication systems with an analytical assessment of the performance is given /6/.

Technical communication systems have to convey messages between several stations and wide use is made of bus systems. Three main actions must be performed for each complete transmission cycle:

1. bus assignment
2. message transmission
3. logical and physical message absorption

These actions can be performed with or without the aid of a central master. In accordance to this we get a classification scheme as shown in Fig. 7. The RDC-system is classified as a DDD-type system.

		MESSAGE TRANSMISSION			
		CENTRAL	DECENTRAL	CENTRAL	DECENTRAL
BUS ASSIGNMENT	CENTRAL	ZZZ	ZDZ	ZZD	ZDD
	DECENTRAL	DZZ	DDZ	DZD	DDD
		CENTRAL I.G. PASSIVELY COUPLED		DECENTRAL I.G. ACTIVELY COUPLED	
		MESSAGE ABSORPTION			

Fig. 7: Classification of bus systems

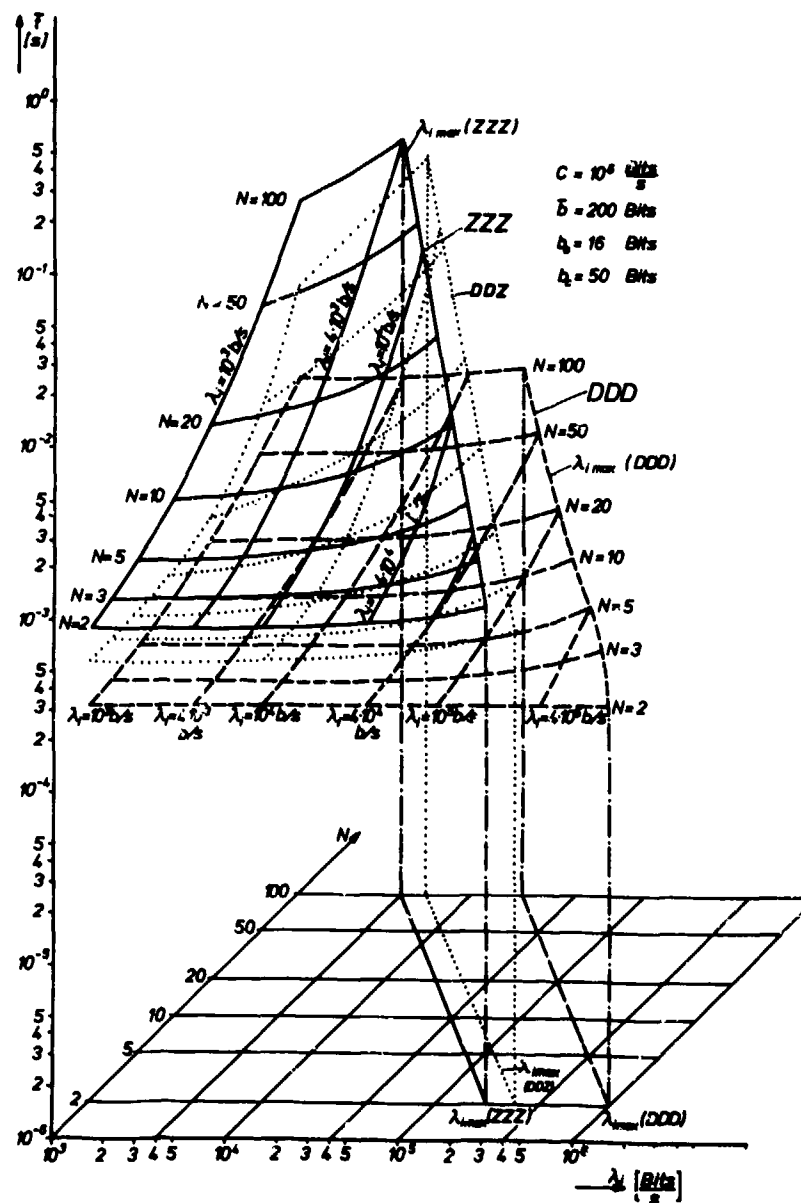
For the analytical assessment two variables are important:

- the average maximum throughput λ_{imax} available for each station and
- the average transmission time \bar{T} from transmission demand in the source station till complete reception of the message in the destination station.

Fig. 8 shows the results for 3 classes: ZZZ, DDZ and DDD. We see three axis with logarithmic scales:

- x-axis: Arrival rate in bits per sec λ_i
- y-axis: Average transmission time \bar{T}
- z-axis: Total number of stations N

The maximum throughput λ_{imax} of DDD is approximately 4 times higher than of a ZZZ system and about 2-3 times higher than of a DDZ system. The average transmission times \bar{T} of ZZZ are higher than DDZ, and DDZ has essentially higher transmission times than DDD in any case.



C = transfer rate, B = message length
 b_0 = word length, b_c = control overhead

Fig. 8: Average transmission time in systems of the classes ZZZ, DDZ and DDD

4. THE TRAFFIC FLOW ON THE RDC RINGBUS SYSTEM IN THE PIT FURNACE APPLICATION

The results gained by analytical means are good for principle and mean value assessments. But the real world is much more complicated as pointed out, for example, by the variables of the workload of a bussystem. There we have a

- distribution of source and destination addresses
- distribution of interarrival time between messages
- distribution of message lengths
- distribution of message types
- frequency of message sequences

All these distributions are locally dependent and dependent in time, with steady-state or unsteady-state behaviour. Moreover, they depend very strongly on the application, particularly the source and destination address distribution. Typical application classes are

- centralized management of distributed control systems (see measurements of the pit furnaces system)
 - pipelined technical process
 - hierarchical structured technical process (tree structure)
 - uniform distribution (totally meshed communication structure)
- and so on.

Because of this complicated world the IITB has developed a measuring processor [7] for gathering the message flow in the RDC ringbus system. But before describing the traffic flow in the RDC communication system and the applied measuring method it is necessary to understand how this system works.

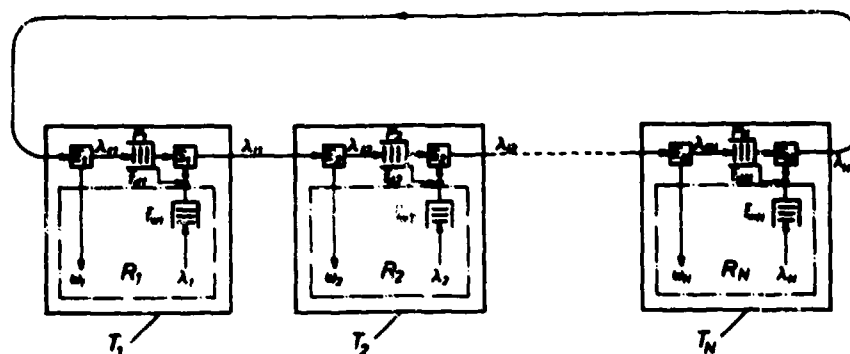


Fig. 9: Queueing model of the RDC-DDD-type communication system

Fig. 9 shows the queueing model of the communication system. The RDC-stations are connected by means of serial fiberoptic lines to a ring structure. Each station has a receiver E_i and a transmitter S_i . In order to achieve communication between the stations each of them has its own address and consequently each message has in its header besides the message type the information where this message (source address) comes from and where this message (destination address) is destined to. Special types of messages (e.g. the status reporting messages) have a broadcast address. Let's have a look on the handling of messages within a station. On receipt of the first word of a message the contained destination address is compared with the station's address by the L_{pi} . If the addresses are equal the arriving message will be delivered to the station and absorbed (ω_i), if it is not equal it will be forwarded to the next station (λ_{fi}). This will be with the minimum delay of one word in all cases in which the queue P_i for passing messages is empty and if the station is not transmitting a message by itself (λ_i). In the latter case the words of the passing message are buffered in the first come first serve queue P_i . Messages of the station (λ_i) are not allowed to be sent if the station is receiving or transmitting a passing message. By this buffer insertion mechanism we have a decentralized bus assignment, a decentralized message transmission and a decentralized message absorption. In this protocol consideration of round-trip delay of the transmission medium is not needed as for instance in collision type

local area networks as ETHERNET.

In order to detect faults on the ringbus system each RDC station has its own fault detectors, and in order to tolerate them there are two fiberoptical rings, one for each direction with the respective receiver and transmitter devices. That means that each direction can be operated unidirectionally or pseudo-bidirectionally in the so-called oscillating mode of operation where the direction is changed periodically.

In the DDD-type communication system a lot of key data are available, but locally distributed. The measuring system was implemented by a further RDC station connected to the serial fiberoptic bus with special functions. These are:

- observation of the traffic flow at its connecting point
- sending and receiving of special measuring messages in order to get the transmission times
- buffering with optional preselection and recording of the measured data.

Because of the knowledge about the different traffic flows in the pit furnace application the optimal connecting point is adjacent to the two computers of the master control room (Fig. 10). Unfortunately, not all distributed traffic flows (e.g. demand messages and the respective reply messages) can be accessed in bus systems of the class DDD at this connecting point, but all critical cases of heavy load can be detected in this way.

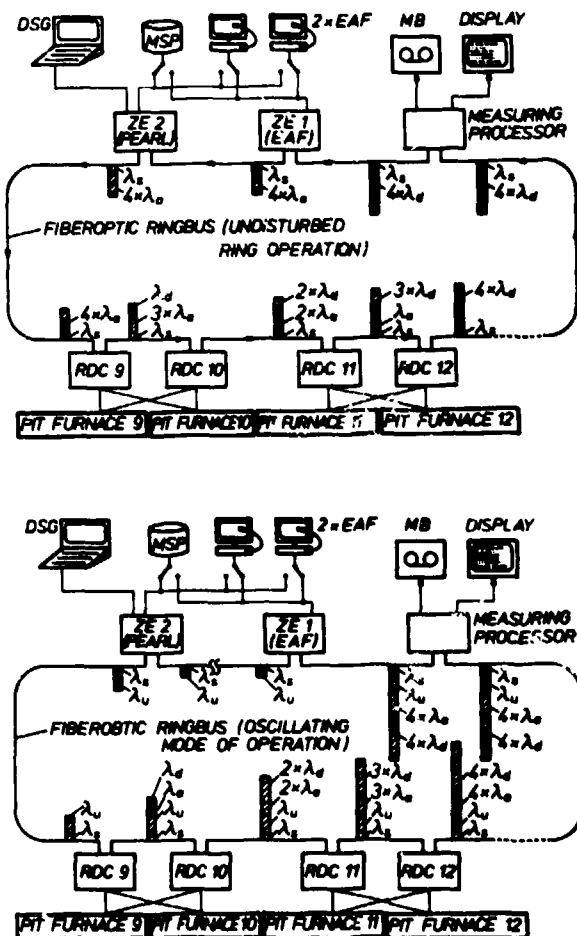


Fig. 10: Traffic flow on the RDC ringbus system in the pit furnace application

In the case of undisturbed ring operation (Fig. 10A) we have three main message types: The status reporting messages with the arrival rate λ_s , the demand messages of the EAF-computer λ_d and the reply messages of the RDC stations with the arrival rate λ_r . In the left ring direction operation mode as shown in this figure all status reporting messages and reply messages pass the measuring system. If we change into the right direction all status reporting messages and demand messages will be accessed. For simulation of the oscillating mode of operation we apply an interruption of the ringbus between the two computers of the master control room (Fig. 10B). In addition to the message types of the undisturbed ring operation we now have the messages with the arrival rate λ_u initialized by the communication computers adjacent to the interruption points to perform the oscillating mode. All messages to and from the EAF computer and the messages of the communication computers pass the measuring system.

At any event, a set of data is recorded with the following contents:

Message type, source address, destination address, message length and clock time (time resolution 10 μ s).

These records are concentrated stepwise and if necessary preselected, recorded on a magnetic tape and finally transferred into a mainframe computer system (Fig. 11).

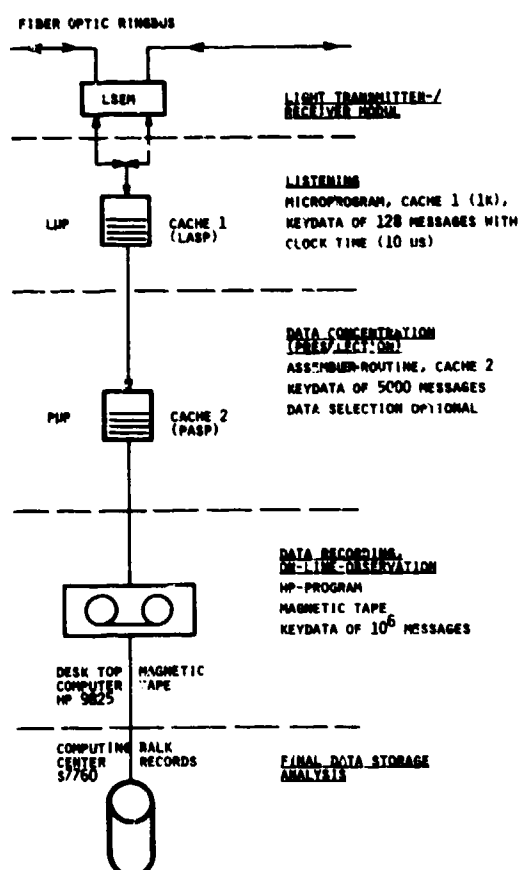
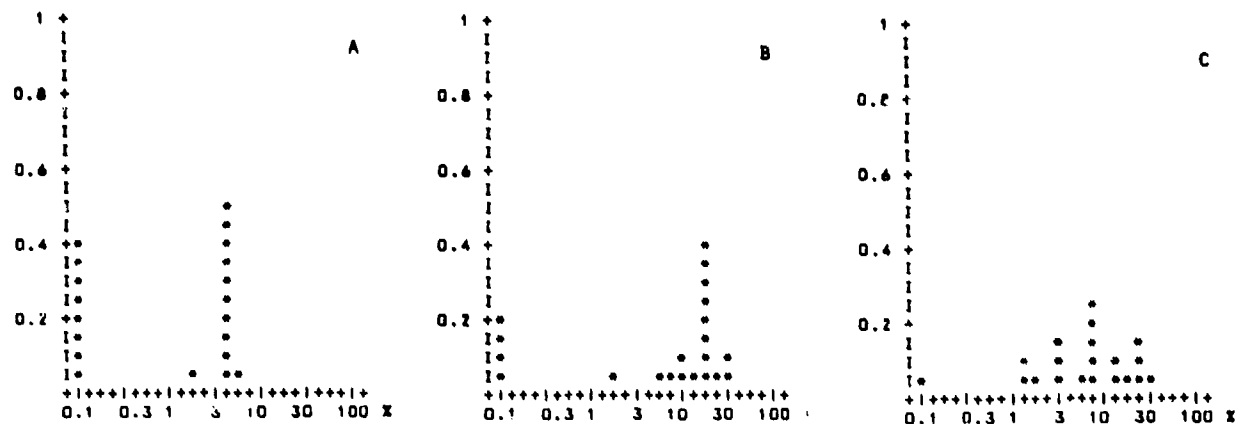


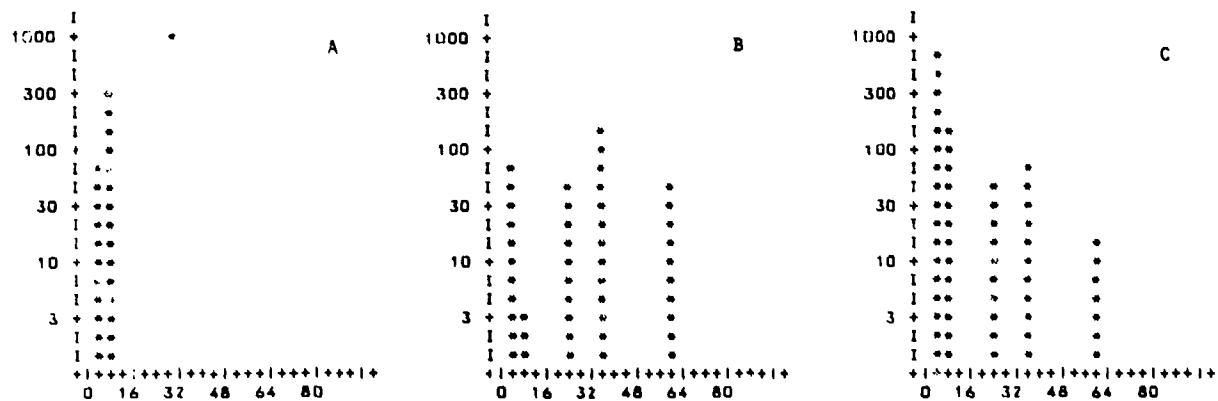
Fig. 11: The measuring and analyzing system

Here, a lot of analyses can be done. Examples for results evaluated from measurements in the pit furnace application are shown in the Figs. 12-14. Fig. 12 shows typical load distributions of the traffic flow for the transmission into the right direction (A) including λ_s and λ_d , into the left direction (B) including λ_s and λ_d and the case of oscillating mode of operation (C), including λ_s , λ_d and λ_u .



y-axis: Frequency; x-axis: Workload in per cent of the transfer rate C

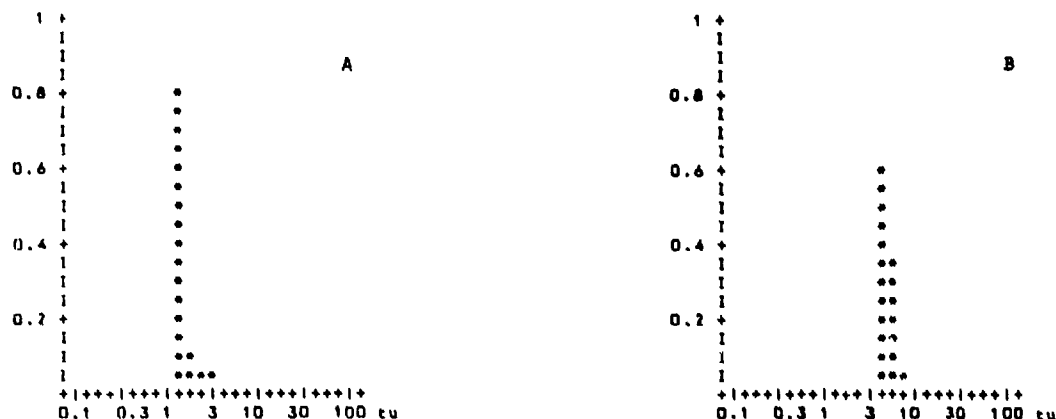
Fig. 12: Typical load distribution of the traffic flow in the pit furnace application



y-axis: Frequency; x-axis: Message length in words

Fig. 13: Distribution of the message lengths corresponding to the diagrams of Fig. 12.

Fig. 14 shows the round trip delay time for test messages of the lengths of 1 data word (A) and of 64 data words (B) respectively.



y-axis: Frequency, x-axis: Transmission time in time units

Fig. 14: Distribution of the transmission times for round trip test messages

In general one sees that all distributions are very peaky in particular in the case of message lengths. The distributions depend very strongly on the application. And therefore, it is necessary to do the classification, the analytical work, simulation work if possible, and measurements, too. The classification gives a framework for thinking; by the analytical work one is forced for a detailed modelling of the general system behaviour, by this one gets an assessment which is good for a general view and for finding instabilities or things like that; by simulating one is able to model more complex models or more special situations which represent, for instance, certain applications; and finally, the measurements with application and artificial patterns are necessary for getting confidence in all these models.

5. REFERENCES

- /1/ Borsl. L., E. Pavlik, 1980, "Konzepte und Strukturen dezentraler Prozeßautomatisierungssysteme", Regelungstechnische Praxis 9 (1980), S.302-309, R.Oldenburger-Verlag München.
- /2/ Syrbe, M., 1981, "The description of fault-tolerant systems". Process automation 1/1981.
- /3/ Heger, D., H. Steusloff, M. Syrbe, 1979, "Echtzeitrechnersystem mit verteilten Mikroprozessoren", BMFT-Forschungsbericht DV 79-01, Datenverarbeitung, April 1979.
- /4/ Heger, D. (Hrsg.), 1981, "Systemergänzungen und Piloterprobung eines fehlertoleranten Echtzeitrechnersystems mit verteilten Mikroprozessoren (RDC-System)", BMFT-Forschungsbericht DV 81- , Datenverarbeitung, Mai 1981 (to be published).
- /5/ Steusloff, H., 1980, "Programming distributed computer systems with higher level languages. Distributed computer control systems, Proceedings of the IFAC workshop, Tampa, Florida, U.S.A., 2-4 Oct. 1979. Pergamon Press, New York.
- /6/ Heger, D., 1979, "Kommunikationsverfahren für Sammelleitungssysteme und deren Leistungsbeschreibung", Regelungstechnik 1979.
- /7/ Heger, D., R. Bähre, 1980, "Meßprozessor für Rekonfigurationsabläufe und Übertragungsströme im Echtzeitrechnersystem mit verteilten Mikroprozessoren (RDC-System)", IITB-Mitteilungen, Karlsruhe, FhG-Berichte, München, S. 2-80.

CONTRIBUTION TO THE DISCUSSION by Mr. J. Schoelch, IABC, Ottobrunn / Germany.

Question: How do you tolerate an interruption on the ringbus system ?

Answer: Let me explain the fault-tolerance by three typical examples

a) Communication computer failure:

Faults within a station are detected by the fault diagnosis unit (BS/SR) or by self test. In case of error the communication computer is disconnected by the bus switch unit (BSU, Fig. 4) and the light receiver / transmitter module (LSEM, Fig. 4) works as a repeater device.

b) Line interruption in one direction (e.g. break of one fiberoptic line):

At any time there are signals on the fiberoptic lines of the current transmission direction, either messages or delimiter bytes between the messages. If a station receives no more signals the receiver / transmitter adaption (SEA, Fig. 4) recognizes this by a time-out and the LuP (Fig. 4) sends a broadcast message to all other LuP's as command for changing the ring direction.

c) Line interruption in both directions (e.g. breakdown of power-supply in a RDC station):

The LuP adjacent to the interruption acts as in case b) but after changing the direction the ring is not closed. This is recognized by the two LuP's adjacent to the interruption point and thus the transmission direction is periodically changed by them. So, in one time period each station can send messages to all other stations on the left and in the following time period to all stations on the right. Thus, the traffic flow is chopped periodically into two directions and the performance is only reduced by the additional messages reversing the transmission direction. If one of the LuP's adjacent to the interrupting point receives signals from a previously failed direction a reconfiguration procedure is started with the re-integration of repaired modules. By this procedure the system reconfigures towards the ring configuration with its full performance.

DISPERSED SENSOR PROCESSING MESH PROJECT

by

Vincent A. Migna
The Charles Stark Draper Laboratory, Inc.
555 Technology Square
Cambridge, Massachusetts 02139
U.S.A.

SUMMARY

The F-8 Dispersed Sensor Processing Mesh (DSPM) project is an exploratory program involved in the development and test of the concept of a network communication structure. The elements of the structure are a Bus Controller and a number of nodes all of which are interconnected by multiple data flow paths. This structure is proposed as the communication medium between the subsystems of a distributed avionic system.

The multiplicity of data paths between nodes, in conjunction with an intelligent controller that constructs, monitors, and controls a virtual data bus composed of these nodes and their interconnecting links, is envisioned as a structure much more tolerant to faults and physical damage than the presently employed avionic data busses.

The virtual data bus constructed by the Bus Controller can be reconfigured by the Controller in response to sensed faults and physical damage; thus, it should be capable of maintaining communication between the various subsystems of an avionics system through more numerous and more severe occurrences of faults and physical damage.

In order to test and establish a data base for this proposed communication structure, the elements that comprise it must be designed and built. These elements are not just the hardware that is inherent to the structure, they also include the algorithms mechanized in the Bus Controller's software, the operating characteristic of the network, and the communication protocol used. The decisions made during the development of the system must be carefully thought out and mechanized in the most efficient and reliable manner possible. This paper addresses the design and associated decisions made during the development of the network hardware and software.

1. INTRODUCTION

The evolution and development of integrated avionic systems in which a number of distributed, dedicated processors perform specific aircraft-control and mission-oriented functions has demonstrated that the communication structure between the elements is the critical hardware element in a distributed computer architecture.

To date, the solution to this problem has concentrated on the means to adapt a bus to such applications. The inherent vulnerability of busses to physical damage and the disabling effects of failed subsystems connected to a bus have required a number of developmental modifications to the basic bus. Multiple busses, redundant busses, cross-strapped busses, and various combinations of these are typical bus modifications. An alternative approach—a network communication structure—has been proposed as a better solution to the problem.

The F-8 Dispersed Sensor Processing Mesh (DSPM) project has integrated a communication network structure to an engineering version of a flight-qualified, triply-redundant, digital flight-control system, the F-8 Digital Fly-By-Wire (F-8 DFBW) System. Operating the communication network in parallel to the existing architecture gives a concrete comparison of dedicated sensor/effector interfaces to the communication network strategy.

Figure 1 displays the layout of the DSPM network. The primary node (Bus Controller) is the central processor. Each other node services one or a small collection of devices that are physically very close together.

2. GENERAL CHARACTERISTICS

Messages flow exclusively from the central processor to the nodes and from the nodes to the central processor. Message routing through explicitly activated links controls the flow of information through the net. Messages are relayed from the central processor and back on a bit-by-bit basis without respect to the intended recipient. The nodes route the information through an assigned reconfigurable conductivity. The Input/Output (I/O) port of each node is always in a listening mode to receive any incoming messages. Configuration commands from the central processor activate particular node ports to transmit messages either back to the central processor or on to other nodes in the network.

The network nodes and links are fully duplex; they can handle incoming and outgoing data simultaneously. The network structure configured by the central processor creates a tree-structured bus between the central processor and the nodes. Because of the multiple paths available from a given node to the central processor, it is believed that this structure will be more reliable and tolerant to faults and physical damage.

The initial design decisions on the DSPM network involved the central processor. The central processor is the most important element to the operation of the communication network. The need for high reliability requires either an extremely reliable processor or redundancy. The fact that the general reliability of processors has not reached the desired level and the relatively low unit cost of processors led us to use triplicate central-processor processing elements.

Experience gained on the triplex F-8 DFBW and other redundant systems has shown that fault-detection and fault masking are greatly enhanced by the synchronous operation of redundant systems. Furthermore, the desire to mechanize a hardware mechanism for the data exchange/compare between processors reinforced this decision and resulted in a design that uses instruction-level synchronization.

The next area of concern involved isolation between the central-processor channels. Each of the processors in the central processor simultaneously executes identical code. Each processor though has unique I/O interfaces. These unique I/O interfaces are mechanized through the use of globally unique I/O addresses that enable each processor to manipulate its own I/O registers without impacting on the other processors' registers, i.e., an instruction that addresses an occupied address in Channel A actually manipulates that register while the same instruction in Channels B and C is a vacant operation.

3. NETWORK HARDWARE

The hardware elements which comprise the communication network consist of a central processor Bus Controller, six nodes, and the interconnecting links. The following paragraphs describe these items.

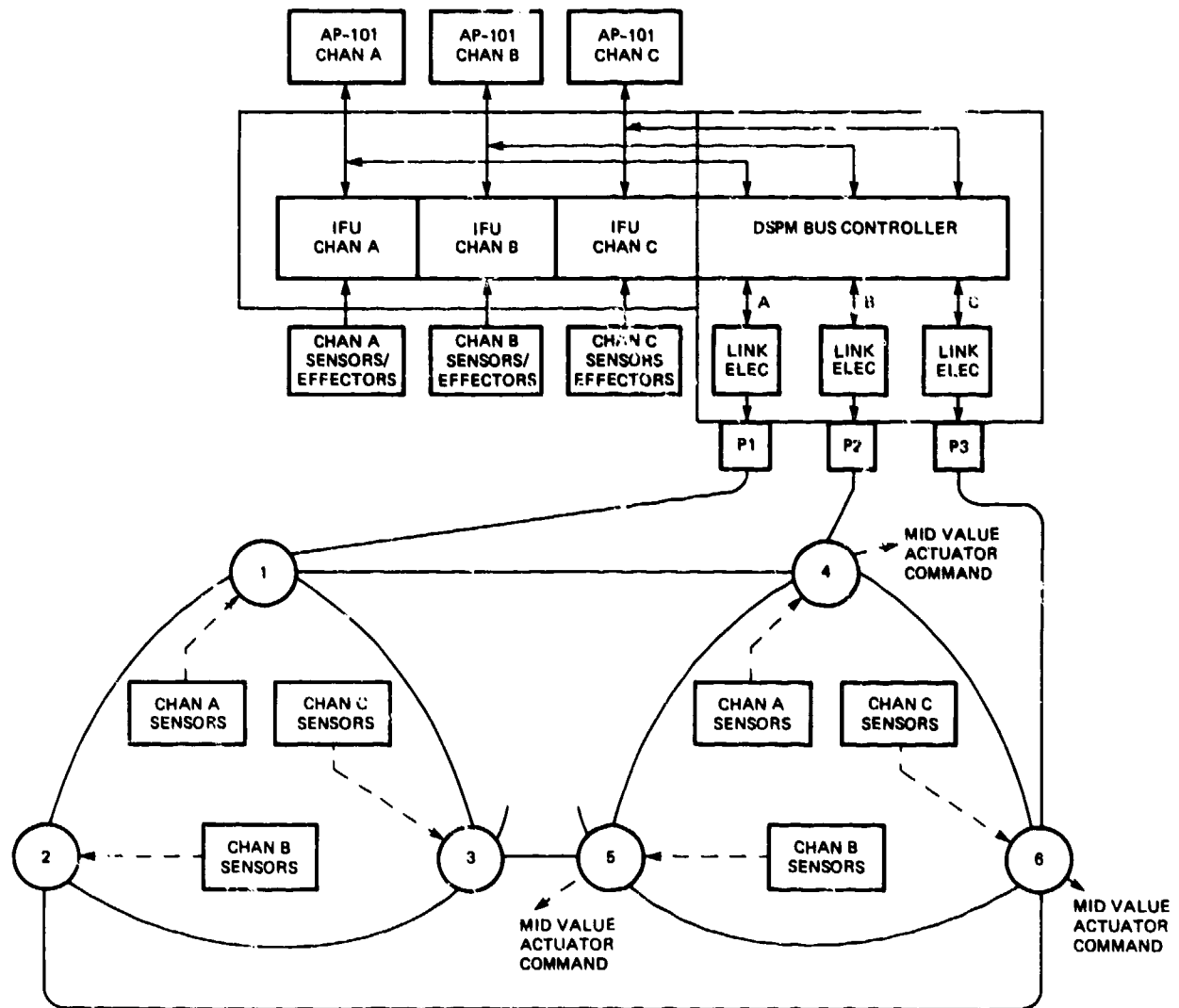


Figure 1. Dispensed sensor processing mesh.

3.1 Primary Node—Bus Controller

The central intelligence of this network resides in the software algorithms mechanized in the Bus Controller. These algorithms manage both the simple message transfers to and from the various nodes and the critical functions relating to the organization and maintenance of the network.

The Bus Controller must configure the nodes to create a suitable network structure, re-establish this structure after power interrupts, detect failed links or nodes, and either rebuild or repair the structure to circumvent the failure. Also, since all links are not in simultaneous use, the inactive links must be activated periodically and used to replace other active links in the structure. This periodic activation maintains up-to-date status information on all parts of the network. The Bus Controller's hardware and software design must incorporate the highest reliability and fault tolerance obtainable to execute these functions.

Each third of the Bus Controller contains a microprocessor, Random Access/Programmable Read Only Memory (RAM/PROM), an Oscillator, Clock Interval Timers (CITs), a Watchdog Timer, I/O interfaces, an Interprocessor Communicator, and an External Interrupt Synchronizer. (See Figure 2.)

Although each section of the Bus Controller operates independently, they are synchronized with each other at the instruction level, and the exchange/comparison of data makes the Bus Controller's redundancy transparent to the network.

3.1.2 Bus Controller Processors

The primary considerations in selecting the microprocessors for the Bus Controller were a 16-bit processor and a large instruction repertory.

Sixteen-bit processor. Experience in programming various computer control systems from spacecraft to ground support equipment has shown that most data manipulation and formatting requires a minimum of 16 bits for both handling ease and data granularity.

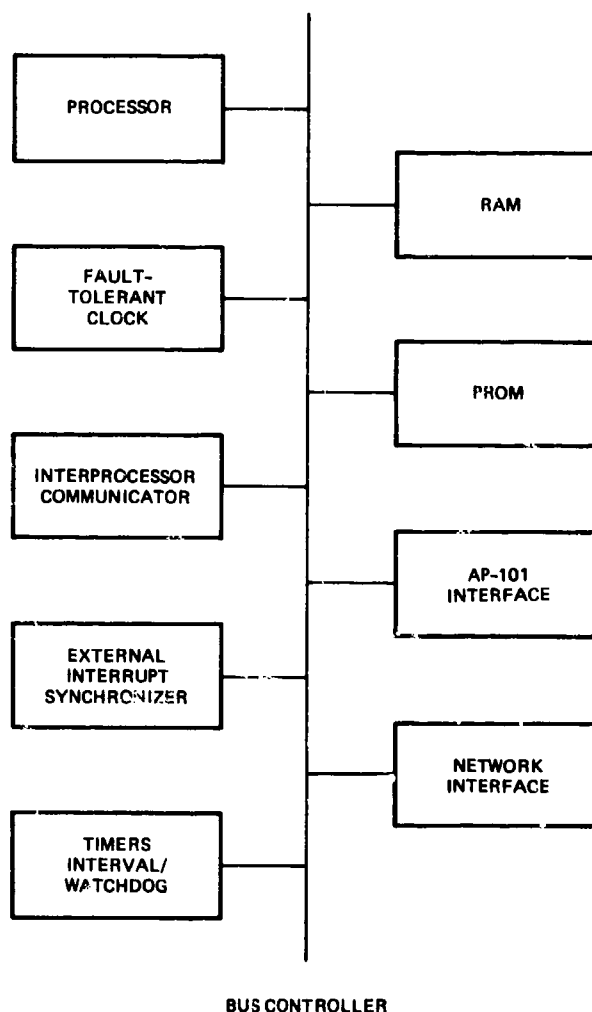


Figure 2. Bus Controller

Large instruction repertory. The programming task eases greatly when the processor has a large instruction repertory. It is much easier to eliminate certain types of instructions from the programming (i.e., exotic addressing modes) than to work with unmechanized instructions.

3.1.3 Bus Controller Memory

The Bus Controller memory compliment involved two main considerations:

- (1) Bits/volume.
- (2) Availability of units.

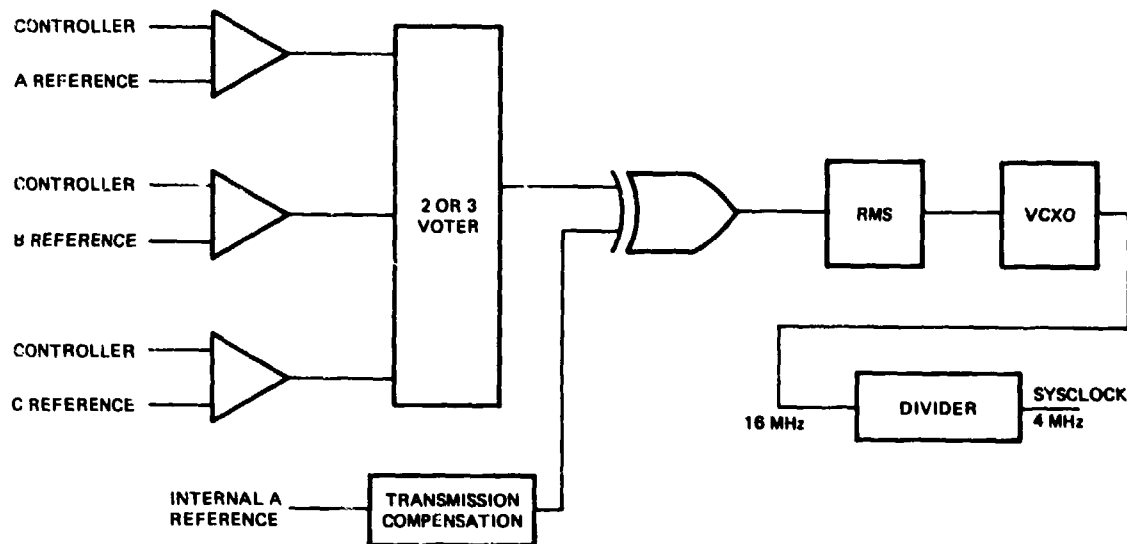
The use of the erasable memory required to develop, test, and modify the operational software was secondary to the design but important to the research aspect of this project.

3.1.4 Bus Controller Clock

Each processor has its own dedicated oscillator as a clock source. The decision to operate at instruction-level synchronization between processors required the active control of these oscillators. The incorporation of a fault-tolerant, phase-locked clock design developed (and subsequently patented) by W. M. Daly and J. F. McKenna of CSDL under NASA sponsorship met this requirement. This clock has a nominal frequency of 16 megahertz with a phase error of 5 degrees (Figure 3). The design is such that the loss of any one oscillator will not inhibit the phase lock between the remaining oscillators.

3.1.5 Interval Timer

The time-cyclic nature of many operational functions in an avionics system and the general need for an interval timer resulted in the incorporation of two 16-bit, 1-microsecond-per-bit interval timers. Experience has shown that this degree of mechanization meets all the expected uses of an interval timer.



BUS CONTROLLER - FAULT-TOLERANT CLOCK

Figure 3. Bus Controller; fault-tolerant clock.

3.1.6 Watchdog Timer

A Watchdog Timer, which requires periodic servicing by system software, is an excellent detector of correct software program execution.

3.1.7 DFBW Computer I/O

The DFBW computer to Bus Controller I/O interface is unique in that the triplex Bus Controller processors act as a single processor but must interface to the DFBW triplex computers as separate units. This is due to the asynchronous relationship between the Bus Controller and DFBW system and the DFBW system frame synchronization versus the Bus Controller instruction-level synchronization.

The complexity of this interface is increased further because the Bus Controller is considered an external device by the DFBW computer, and the external device must respond with the correct handshaking protocol during I/O execution and initiate buffered I/O execution. In fact, this interface must function for both direct output, which is a DFBW computer macro instruction execution, and buffered I/O, which is device initiated and executes in the DFBW computer on a cycle-steal basis transparent to software program execution. The direct output execution time is approximately 10 microseconds to transfer two 16-bit words across this interface, whereas the buffered I/O execution time is approximately 15 microseconds per 16-bit word transferred across the same interface. In order to meet these diverse requirements, this interface is mechanized with two first-in, first-out (FIFO) buffers; control logic; and a FIFO status register between each DFBW computer and Bus Controller processor (Figure 4). Communication between this interface and the Bus Controller processor is on an interrupt basis for information flow from the DFBW computer to the Bus Controller and on program control for information flow from the Bus Controller to the DFBW computer.

3.1.8 Network I/O Interface

Three ports on the Bus Controller connect to the communication network. Each port is uniquely associated with a processor.

The interposition of a bus between the controller processors and the three ports such that any processor could address any port would require either I/O serialization between processors and ports or the interconnection of a minimum of 17 lines between each processor and each port. The need to interconnect this number of wires would create a packaging problem and a possible multiple source of failures that could affect reliability. Serialization of this processor/port interconnect would impose a transmission-time penalty on I/O. Therefore, each processor is uniquely associated with an I/O port, and the use of globally unique addresses for I/O registers allows a single processor to manipulate a port without impact on other ports.

Each port is a 1-megahertz serial connection into the communication network. This serial connection could be mechanized with a number of different protocols. The protocols examined were: Synchronous Data Link Control (SDLC), Asynchronous Digital Data Link, and MIL-STD-1553. Although the first two of these methods would be easier to mechanize from both an operational and a hardware point of view, the MIL-STD-1553 was chosen for its compatibility with a wide range of presently operating avionics subsystems (Figure 5).

The 1553 interface actually mechanized in the Bus Controller and each node is modified to conform to the network operating characteristics. Communications between the Bus

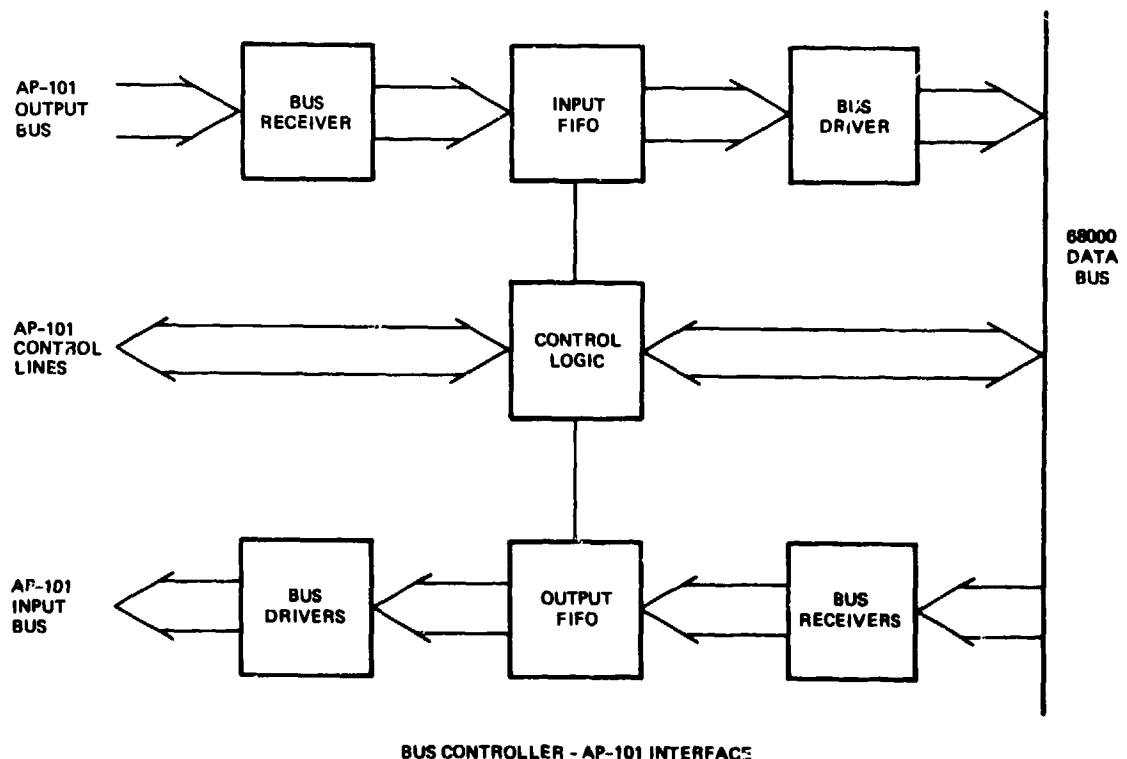
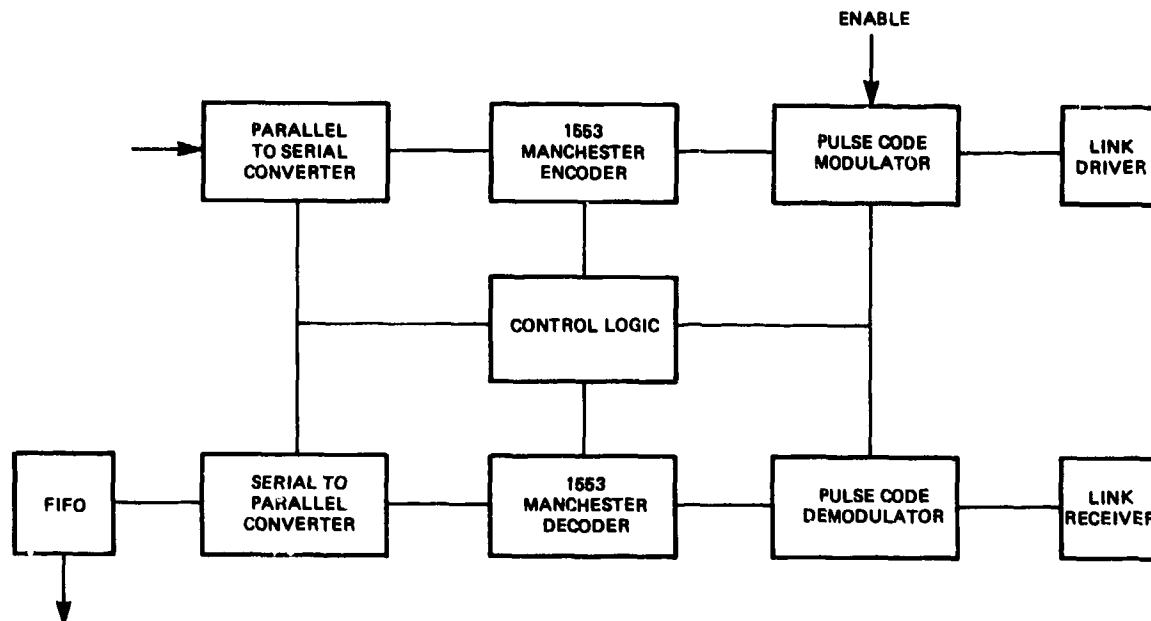


Figure 4. Bus Controller; AP-101 interface.



BUS CONTROLLER - NETWORK INTERFACE

Figure 5. Bus Controller; network interface.

Controller and the addressed node are received and retransmitted by each node between the Bus Controller and the addressed node. Transmission of a standard 1553 pulse-code-modulated Manchester code would suffer both distortion and time delay by this receive/transmit function. Therefore, the 1553 interface in the Bus Controller and each node is modified such that the Manchester code is converted to a self-clocking pulse-width-modulated code for transmission on the interconnecting link, converted back to Manchester code within each node for decode, and reconverted to self-clocking pulse-width-modulated code for retransmission (Figures 5 and 6). Thus, the actual 1553 interface can exist between the node and its attached device but not between the Bus Controller and the device.

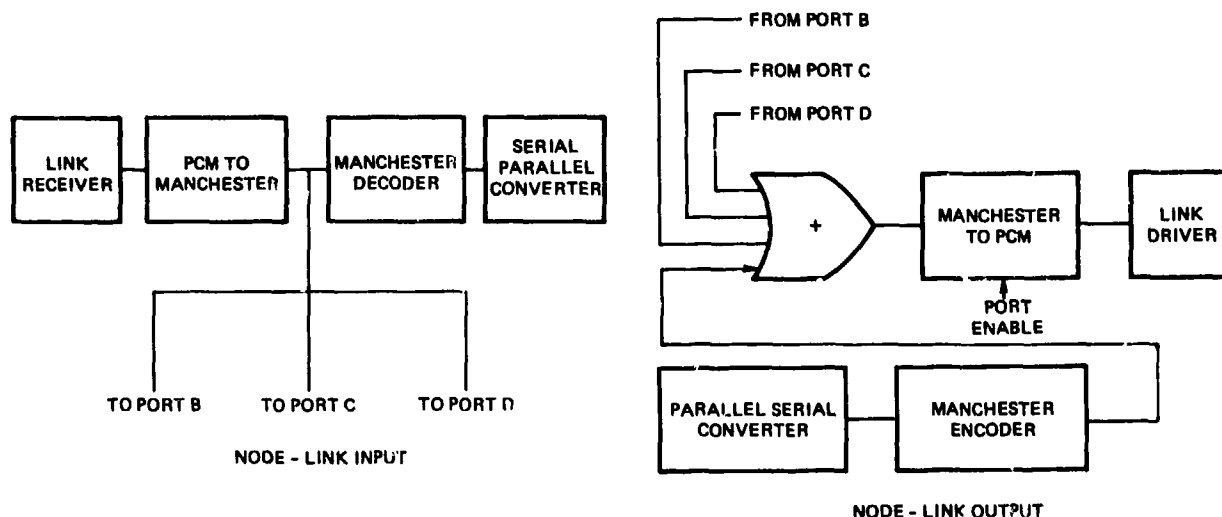
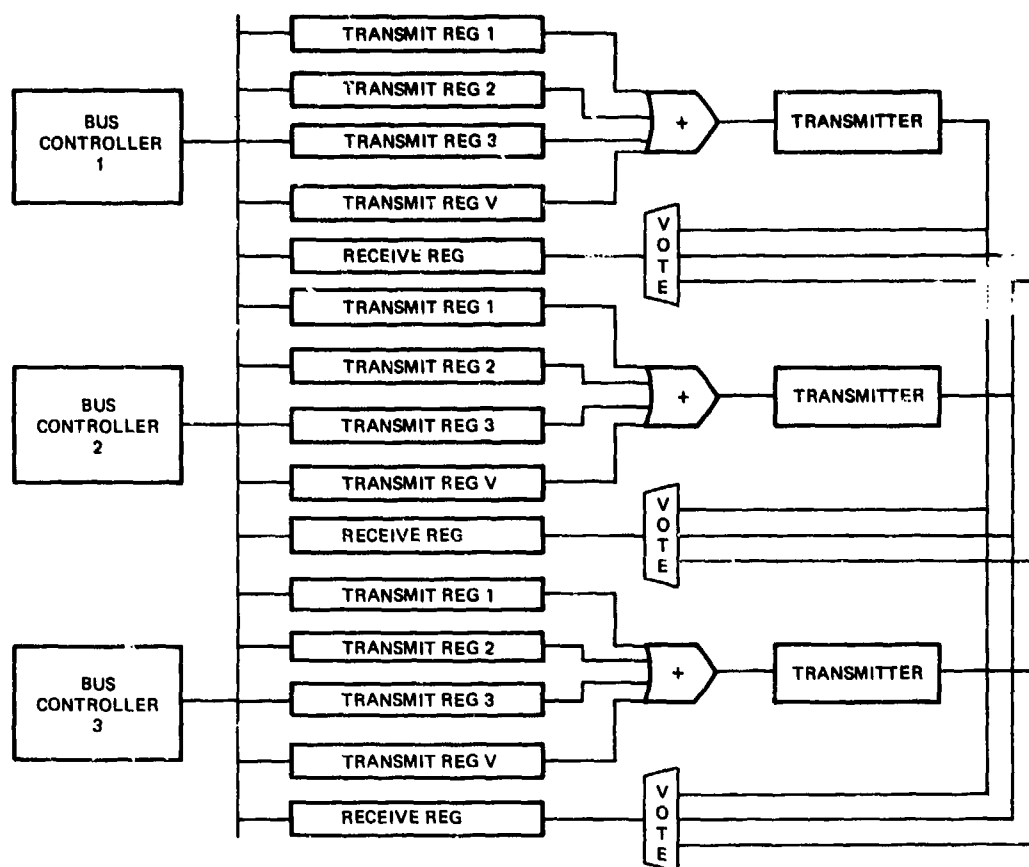


Figure 6(a). Node; link input.

Figure 6(b). Node; link output.

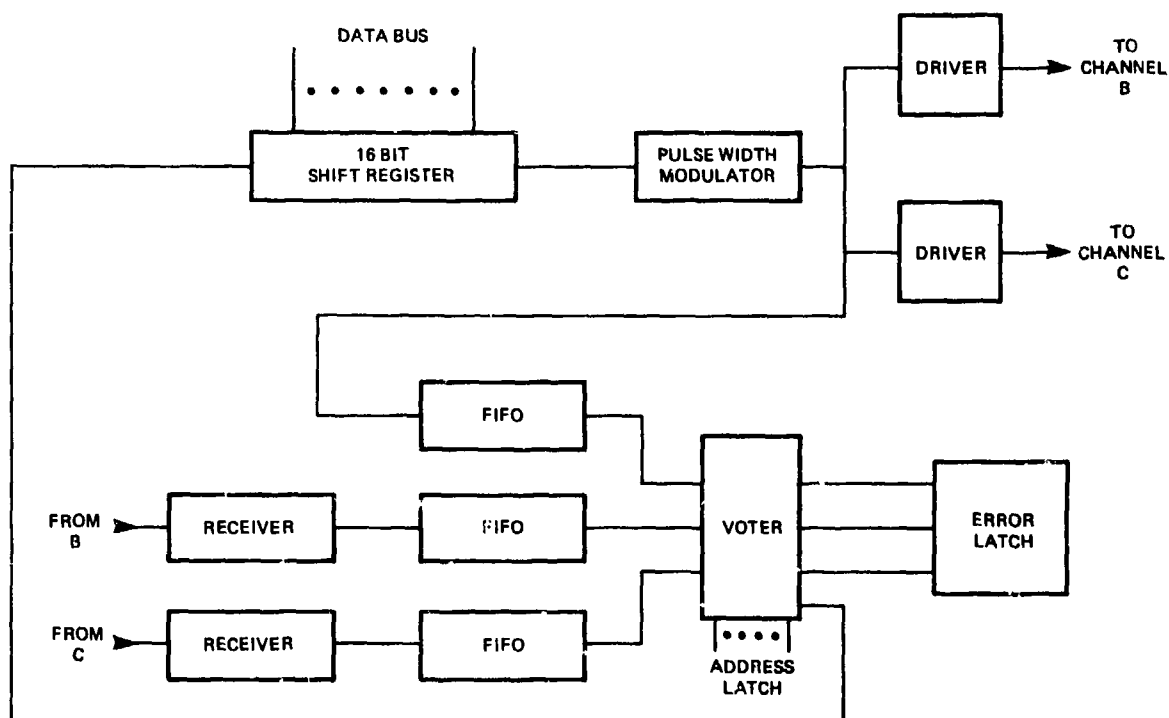
3.2 Interprocessor Communicator

The exchange and comparison of all I/O data between processors is essential to failure detection and fault masking in a multiprocessor controller. Various combinations of software and hardware logic can be chosen as the method through which the exchange/comparison of data can be mechanized. For the DSPM, the design uses all hardware logic, and the Interprocessor Communicator is the mechanism that executes this function (Figures 7 and 8).



INTERPROCESSOR COMMUNICATOR

Figure 7. Interprocessor communicator.



INTERPROCESSOR COMMUNICATOR

Figure 8. Interprocessor communicator.

The Interprocessor Communicator is used to distribute data from one channel to all other channels and to exchange data simultaneously from all channels with comparison, error flagging, and correction. The data exchange mechanism only functions correctly between synchronized channels. An attempt to exchange data between unsynchronized channels will generally cause a loss of the transmitted information (the transmitted information is not correctly received by anyone except the transmitter himself). An attempt to accept a data value from an unsynchronized channel produces an indeterminate result, most often a zero. Table 1 summarizes the functional registers of the Interprocessor Communicator.

Table 1. Interprocessor communicator registers.

Transmitter Registers	
Name	Function
XV	Transmit a single data item from all channels. The received value will be the bit-by-bit 2-of-3 majority function. A bit fault will set the corresponding bit of the error latch.
X1	Transmit a single data item from Channel 1 to all channels. The received value in all channels will be that value. The error latch status is unaffected.
X2	Transmit a single data item from Channel 2 to all channels. The received value in all channels will be that value. The error latch status is unaffected.
X3	Transmit a single data item from Channel 3 to all channels. The received value in all channels will be that value. The error latch status is unaffected.
Receive Register	
XR	The received result of a transmitted operation can be retrieved from this register. The register is read/write and can be directly loaded by a store into the register.
Status Register	
XE	<p>A status register that contains 4 bits.</p> <p>Bit 0 is an error bit corresponding to Channel 1.</p> <p>Bit 1 is an error bit corresponding to Channel 2.</p> <p>Bit 2 is an error bit corresponding to Channel 3.</p> <p>Bits 3-14 are always zero.</p> <p>Bit 15 is a summary bit that is set if any of the bits 0 thru 3 are set and is clear if they are all clear.</p> <p>The register bits are set if a bit fault is noted for the corresponding channel during an XV, operation.</p> <p>The register is read/write and can also be directly loaded by a store to the register. Only bits 0 thru 2 (and by function bit 15) can be altered by a store to the register.</p>

A store to a transmit register and a subsequent fetch from the receive register, XR, exchanges data between channels. The transmit register selection determines the character of the exchange. All channels must execute the store to the exchange register in synchronism. The error latch, XE, is set if errors are detected during XV exchanges. Once an exchange has been initiated by a store to an exchange register, the processor proceeds with normal instruction execution. While a transfer is in progress, a reference to the exchange mechanism suspends the processor instruction execution until completion of the exchange. An exchange operation takes 3 microseconds. An immediate reference to the receive register, XR, after a store is thus likely to suspend operation of the processor for about 1 microsecond to allow time to complete the exchange. Since the interlock mechanism is automatic and the maximum loss of processor time per exchange is negligible, the programmer need not concern himself with either assuring completion of the exchange before accessing the received data or padding the program between exchange requests and subsequent receiver register, XR, accessing.

This design was chosen because the exchange/comparison of data takes minimal execution time, requires instruction-level synchronization between processors, and avoids the need to establish criteria for agreement between two nearly identical results. Agreement is, by definition, bit-for-bit.

Through the use of globally unique I/O addresses, the various Interprocessor Communicator transmitter registers are mapped to different memory locations for each Bus Controller processor. Thus, only Channel 1 can manipulate the X1 transmit register while the execution of an identical instruction in Channels 2 and 3 is a vacant operation. However, the instruction execution sets the address latch in the communicator which then passes the correct output to the receiver.

When all processors simultaneously address the XV transmitter, a bit-for-bit comparison of the inserted data is performed. The voted results are passed to the receiver, XR, and XE records any discrepancy. The processors can then use the XE information for fault detection.

3.3 External Interrupt Synchronizer

The instruction-level synchronization between processors imposed the requirement that all processors process external interrupts simultaneously even though the external device generating the interrupt was associated with only one processor.

Devices requiring service signal the event through the generation of an interrupt. Since these interrupts can occur at random times in each channel, and, if processed randomly would destroy the instruction level synchronization between processors, an Interrupt Handler is positioned between the devices and the processor (Figure 9).

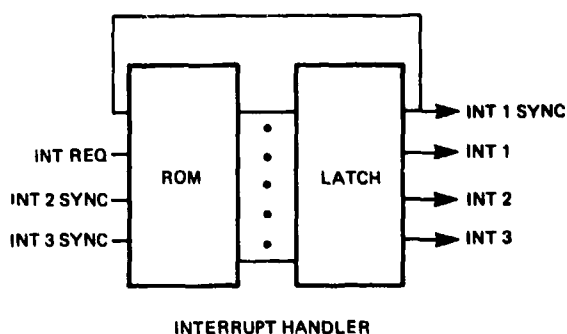


Figure 9. Interrupt handler.

The function of the Interrupt Handler is to record the identity of the interrupting device and transmit the fact that an interrupt is pending to the Interrupt Handlers associated with the other processors. When all Interrupt Handlers have been notified, an interrupt is simultaneously applied to each processor. The interrupt that actually triggers interrupt processing in the processors is tagged as originating with some actual channel device (i.e., interrupts from Channel 1 devices will, through the Interrupt Handler, generate a level 1 interrupt, while Channels 2 and 3 will generate level 2 and level 3 interrupts, respectively). The interrupt processing routine that begins execution will, therefore, only address the actual device in the processor responsible to service that device. In its simplest form, each processor has three routines to service a device, and the particular routine executed depends upon the interrupt level seen by the processor.

Use of the Interprocessor Communicator makes the singularity of the processor/device relationship transparent to the processors. The routine that executes in the processors allows only one processor to manipulate the device register, but all processors receive the data from the actual interrupting device through the Interprocessor Communicator which has an addressed transmit register that is keyed to the interrupt level.

The identity of the actual interrupting device is supplied to the processor in an Interrupt Status Word extracted from the Interrupt Handler when interrupt processing begins. The various channel devices will set a bit in this word to indicate service required. This bit is cleared when the device has been serviced. Figure 10 is an example of the software use of the Interprocessor Communicator in response to inputs from the External Interrupt Handler.

4. NETWORK MESSAGES

The information flow in the network consists of Command messages and Response messages. Command messages are at least two words long and consist of a command word and one or more data words. Response messages are either a single status word or a number of data words.

The Bus Controller transmits Command messages into the network of the form shown in Figure 11 and Table 2. Command messages have two formats: Node directed messages and Device directed messages.

4.1 Node Directed Messages

Node directed messages consist of a Command word that identifies the message and a data word that defines the node ports enable/disable configuration.

4.2 Device Directed Messages

Device directed messages consist of a Command word that identifies the message and one or more data words. The most significant bits (MSB) of the first data word following the command word signifies whether it is a true data word for the device or a word that defines some action the device is to perform.

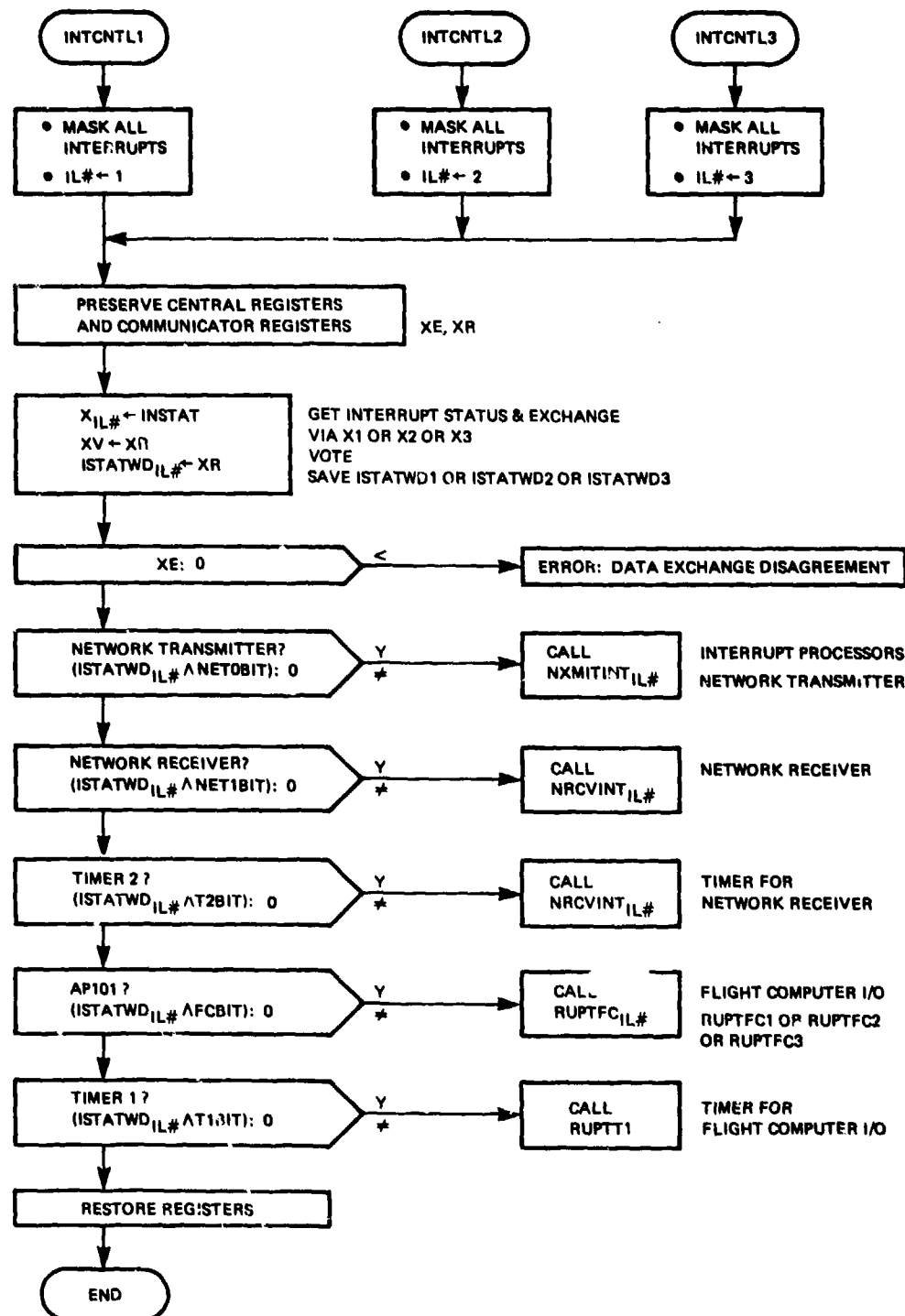


Figure 10. Interrupt controller (interrupt levels).

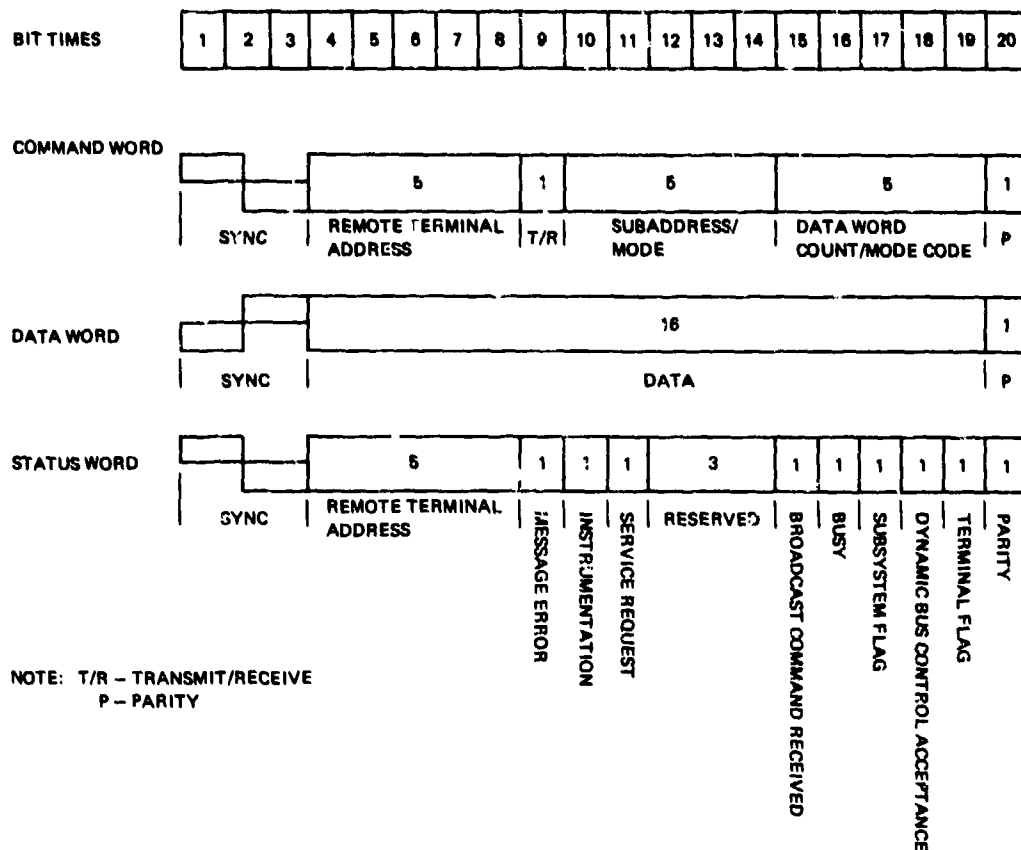


Figure 11. Word formats.

Table 2. Command message format.

Command Messages	
Command Word	
Remote Terminal Address	Definition
11111	Message to Node
00000	Reserved
00001	Message to Device 1
00010	Message to Device 2
etc.	
Subaddress/Mode	
NNNNN	Individual Node Address
Data Word Count/Mode Code	
00001	Number of Data Words Following Command Word
00010	
etc.	
T/R Bit	Always set to 0
Data Word	
1XXXXXXXXXXXXAA	Node Port Enable/Disable
0DDDDDDDDDDDDDD	Data or Command to Device
0000DDDDDDDDDD	Device Data
0111CCCCCCCCCCC	Device Command

4.3 Node Response Message

The Response Message issued by a node to acknowledge the receipt of a Node Directed Message consists of a single status word as defined in Figure 11. This response is automatically executed by the node hardware logic.

4.4 Device Response Message

The Response Message issued by a device to acknowledge the receipt of a Device Directed Message is device dependent and message dependent. The response might be a single status word signifying message received, or it could be a number of data words in response to a data request.

4.5 Intramessage Word Gap

The time gap between the words that compose the Command Messages is on the order of 5 microseconds, while the gap between multiple words in a Response Message is device dependent.

5. NODE HARDWARE

Each node is logically and physically divided into two sections—a Communication section and a Device Servicer. The Communication section executes all functions that relate to the activation/deactivation of I/O ports, receipt and retransmission of all messages, and recognition of messages directed to its attached device (Figure 12).

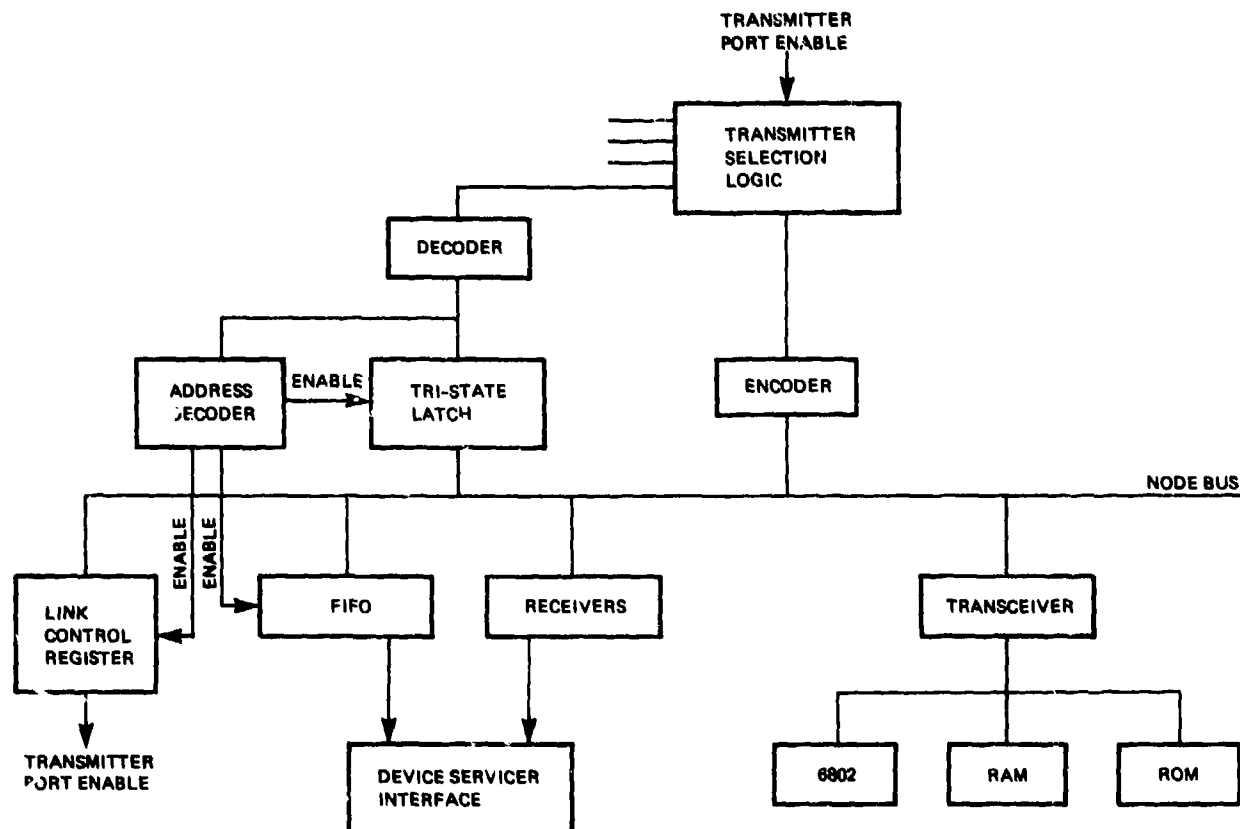


Figure 12. Node detail.

5.1 Node Communication Section

The desire for minimal time-loss impact on the receipt and retransmission of messages and on the execution/response to port configuration commands dictated the design of this section. Therefore, each node has a dual message-flow path. One path receives and retransmits all messages without any examination, and the second simultaneously examines each message for the identity of its appropriate recipient.

Node configuration commands are executed by hardware logic within the Communication Section, and an immediate response to receipt of command is transmitted to the Bus Controller. Messages directed to the attached device are passed to the Device Servicer section which transfers the message to the device. In this case, the response is not necessarily immediate, as it is a received-message function.

5.2 Device Servicer Section

The Device Servicer Section consists of a FIFO buffer, receivers, and control logic between the node bus and the attached device. This design enables the node to recognize messages directed to the attached device, temporarily store them, and then transfer the message to the device at the device acceptance rate. Therefore, the device does not need to process messages at the network data rate. Data flow from the device to the node for transmission to the Bus Controller is executed at the device rate.

5.3 Attached Device

Devices that are attached to nodes can have many forms and functions. Figure 13 defines one of the devices used in the DSPM network.

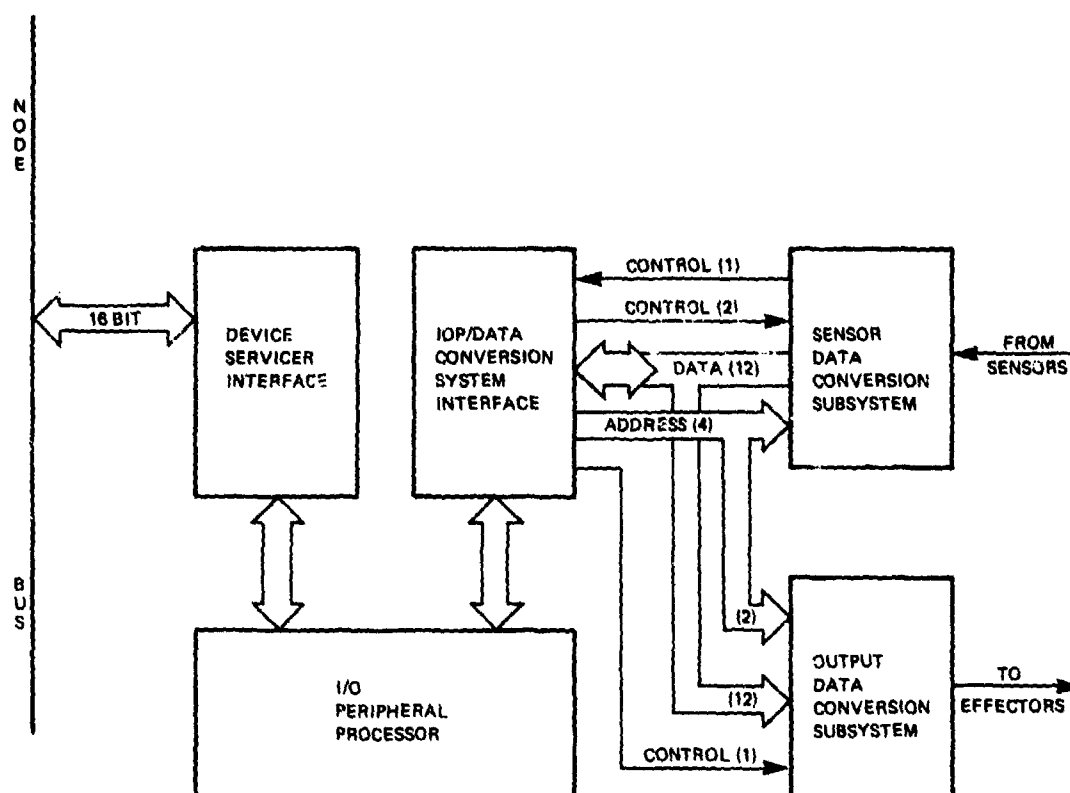


Figure 13. Device servicer.

5.3.1 Node Processor

The node Communication section is designed to use hardware logic while executing all of its functions. In order to facilitate both the expansion of node capability and node test, a microprocessor with associated memory was inserted into the design. This processor interfaces directly to the node bus, and therefore, it can be used to execute more detailed types of node configuration commands and/or introduce controlled types of failures into the network structure, i.e., failed link, babbling node, scrambled messages.

6. DSPM SOFTWARE

The Bus Controller and network structure function directly influenced software design and development. This function is the collection and dispatch of data between sensors, DFBW computers, and aircraft effectors. Essentially, the Bus Controller is an interrupt-driven processor that executes tasks directed by the DFBW computers.

6.1 Software Structure

The software structure is a list of event-dependent tasks. Either an interrupt occurrence or a fault detection signals an event. Each task (or response to an event) has a priority that determines its sequence of execution. Network fault-detection tasks have the highest priority since no other task can be accomplished while the network is not functional.

6.2 Code Generation

The use of a general programming language (PASCAL) was investigated early in the software development phase. A number of routines were coded in PASCAL and the assembly-level language. The memory space and associated execution time required by PASCAL-generated code was too inefficient and slow; therefore, all coding was performed in assembly-level language.

6.3 Support Software

Support software for the selected microprocessor, and most microprocessors, suffers from a prime deficiency. The program must be assembled and the image produced from a single file. This creates problems in code modularization and the use of mnemonics. Individual programmers must be careful in their selection and use of mnemonics in order to prevent dual use and library overflow. Also, module boundaries lose their identity.

The solution to these support software deficiencies was to divide the program into sections, each of which starts at some absolute address. Then, the program sections were linked together through a common data section that contains "jump tables" and other necessary linkage devices for intermodule communications.

7. CONCLUSION

A network structure's effectiveness in maintaining communications between the various elements of a total aircraft avionics system can only be demonstrated by actually building and testing such a structure. This structure must be exposed to the most comprehensive possible test matrix of simulated physical and fault damages. The structure's response to these imposed failures must be measured for both the level and the speed of response. The degree of data loss during communication, the data bandwidth, and the duty cycle must be measured under various operating conditions. Only when a comprehensive data base, established through the operation of such a communication network, has been compiled can an intelligent decision be made on the effectiveness of this communication strategy.

BIBLIOGRAPHY

1. Cattel, J.J., and Kemp, A.M., Damage and Fault-Tolerant Network Incorporation Into F8 Digital Fly-By-Wire System, CSDL Report R-1309, Cambridge, Mass., August 1979.
2. Hopkins, A.L., and Smith, T.B., "OSIPIS-A Distributed Fault-Tolerant Control System", Digest, 14th IEEE Computer Society Int. Conf., San Francisco, Calif., March 1977.
3. McKenna, J.F., Demonstration and Evaluation of a Fault-Tolerant Input/Output Network, CSDL Report R-918, Cambridge, Mass., September 1975.
4. Smith, T.B., A Highly Modular Fault-Tolerant Computer System, Ph.D. Dissertation, Dept. of Aeronautics and Astronautics, MIT, Cambridge, Mass., November 1973.
5. Smith T.B., "A Damage-and-Fault-Tolerant Input/Output Network", IEEE Transactions on Computers, Vol, C-24, No. 5, May 1975.
6. Szalai, K.J., and Megna, V.A., "Development of a Multicomputer Fault-Tolerant Digital Fly-By-Wire System", Third USA-Japan Computer Conference, San Francisco, Calif., October 1978.

NEXT GENERATION MILITARY AIRCRAFT WILL REQUIRE HIERARCHICAL/MULTILEVEL INFORMATION TRANSFER SYSTEMS

James W. McCuen
Hughes Aircraft Company
Fullerton, California, U.S.A.
TP 81-16-2

ABSTRACT

Changes in avionic subsystems and mission roles of next generation aircraft will require new concepts in data transfer. New aircraft will need total airframe/weapon system integration which means new approaches must be developed for the interconnection of avionic subsystems. Effort has begun to develop a Military Standard (MIL-STD) which will define the requirements for a high speed data bus network. The SAE/A-2K Subcommittee on Multiplexing has accepted the task of developing this MIL-STD. The standard shall characterize a higher order Information Transfer System (ITS) that will interconnect avionic systems, that contain their own multiplex ITS, into a fully integrated data complex. The higher order ITS shall employ an operational protocol that will provide subsystems and common sensors, independence and fault isolation by distributed control of the common data bus.

This paper presents an overview of the functions to be considered in developing the standard. Several ITS architectural configurations are presented to show bus network topology and data transfer path requirements at the subsystem black box level and at the aircraft/mission level.

INTRODUCTION

Future advancements in aircraft basic flight and weapon subsystems accompanied by the need for total avionics system integration will demand changes in both intra and inter subsystem data transfer as we know it today. These changes are due to many factors, some of which are:

- Need to eliminate costly hardware/software elements required of centralized controlled, data transfer systems.
- Dispersion of microprocessors within subsystems necessitating the interchange of processed data between subsystems.
- Need for the generation of an aircraft data base, available to all subsystems, which includes all airframe/mission parameters.
- Maximizing the use of common sensor data.
- Making maximum use of multifunctional Control/Display (C/D) elements.
- Maximizing the use of common sensor data.
- Making maximum use of multifunctional control/Display (C/D) elements.
- Allowance for further standardization of hardware/software elements by use of other MIL-STD's, e.g., 1553, 1589, 1750, and 1760 for interchangeability between weapon systems and aircraft.

Present day military aircraft employ only single level-centralized controlled, command response type, information transfer systems. Those aircraft with multiple ITS which require interchange of data, communicate with one another via mutually supported, memory storage interface units. With subsystems integrated in this manner, a change in one results in unpleasant ripple effects progressing throughout the others. This action is due to the changes necessary in the centralized command/response software packages in each of the supporting ITSs.

A solution to this problem is the development and use of an ITS which will efficiently interconnect in a hierarchical order, multilevel multiplexed ITSs. With such an approach a higher order operating system (Mission Management) can be created which provides the processing of functions required of multi subsystem inputs. Such a high speed Higher Order Transfer (HOX) system, employing contention protocol will provide each lower level ITS a functionally isolated communication medium whenever data interchange is required.

The extensive use of MIL-STD-1553 bus networks has proven the concept of multiplexed data transfer systems to achieve a degree of integration. Unfortunately 1553B protocol does not provide the characteristics (speed and protocol) needed to efficiently operate with future hierarchical/multilevel networks. MIL-STD-1553B characteristics are ideally matched to many intra avionics subsystem data transfer requirements which necessitates sensor data collection, central processing, then distribution of results to peripheral areas, e.g., Electrical Power Control, Flight Control, Propulsion and Stores Management subsystems. There will be continued use of 1553B bus networks for intra subsystem data transfer.

In the next decade we can expect some well known subsystems to be combined and the appearance of new ones. Each major subsystem will have its own intra multiplexed bus network. All these asynchronous operating ITS will need to be interconnected to create an integrated data base. Such a data base will maximize use of common data and allow for continuing changes in the subsystems and total airframe/weapon tasks with minimum disturbance to the higher order ITS. Characteristics of the HOX system must provide for isolation of flight critical functions, allow for independent design, production and test of subsystems and incorporate distributed bus control, to eliminate costly central hardware/software complexes.

HIERARCHICAL ITS ARCHITECTURE

The composition of an avionic subsystem suite and the aircraft's mission role will determine the configuration of the higher order ITS that should be used. Examples of hierarchical/multilevel ITS configurations are shown in Figures 31-1 and 31-2 to illustrate different ITS topology. Figure 31-1 shows this architectural thinking by incorporating three levels of ITS to interconnect the full complement of avionic subsystems and units shown. These levels, being identified as bus networks, operate under their own control independent of other bus networks.

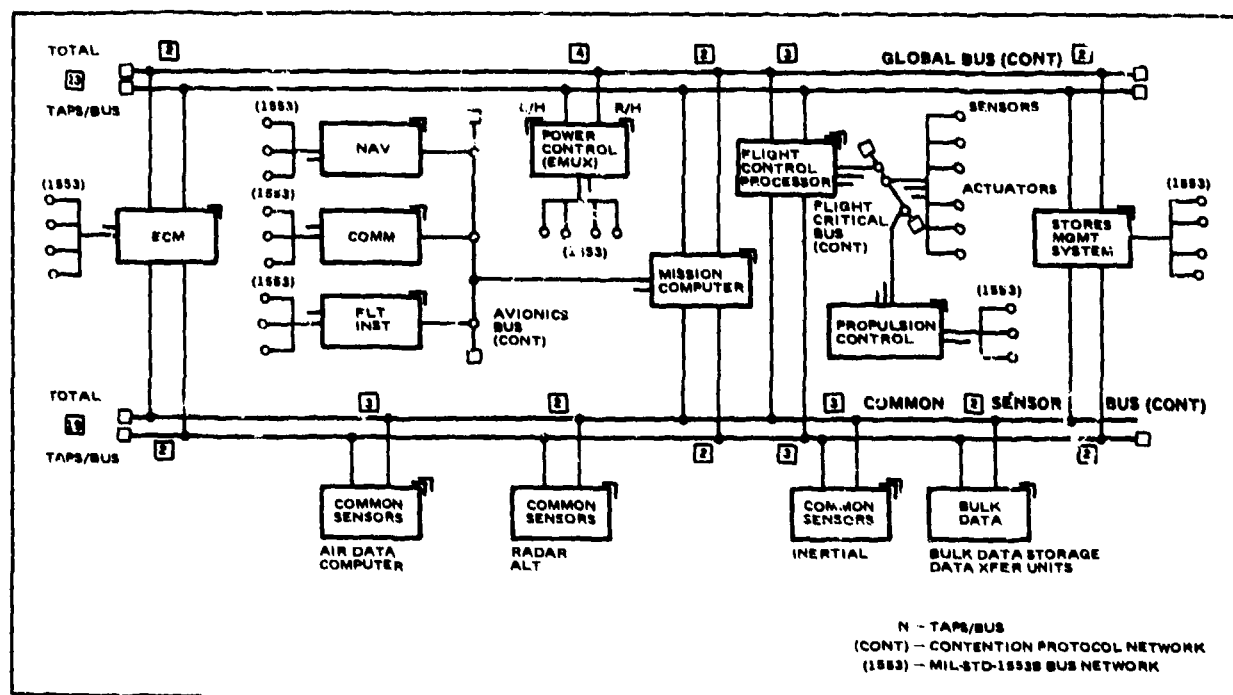


Figure 31-1. Three Level Information Transfer System (ITS) with Dual-Higher Order Transfer Networks

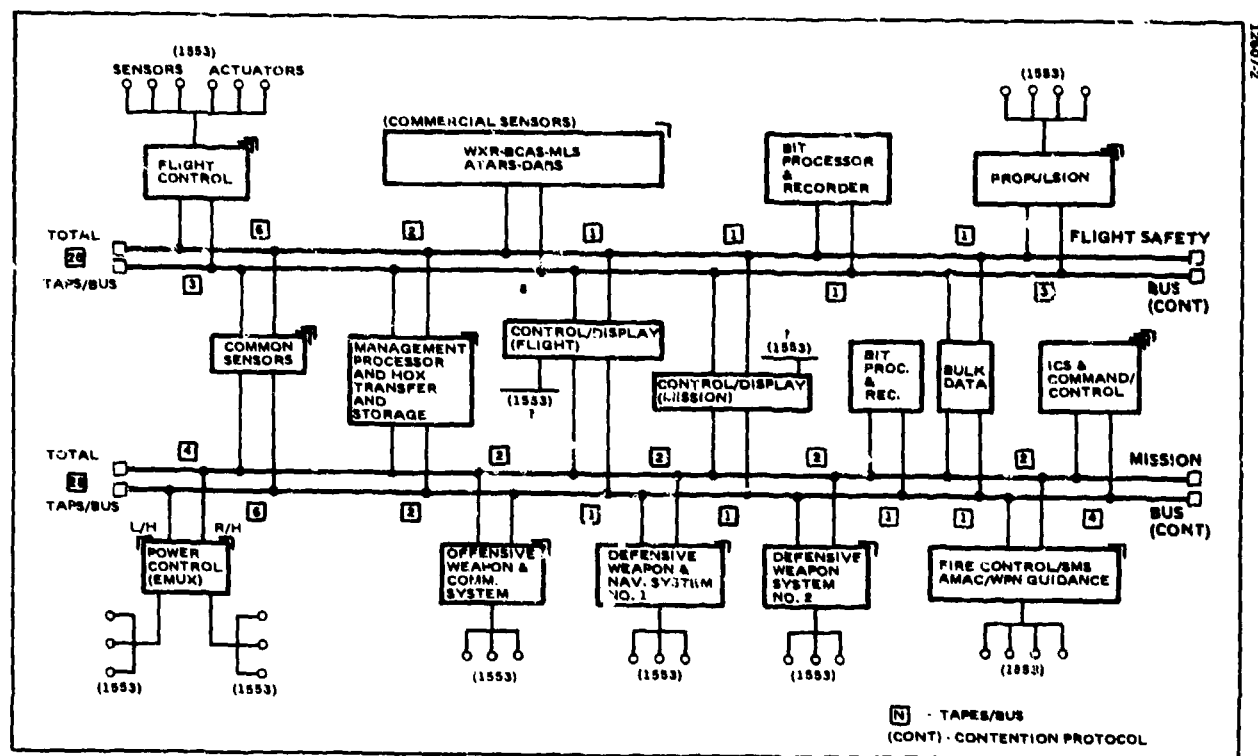


Figure 31-2. Two Level Information Transfer System (ITS) with Dual-Higher Order Transfer Networks

Operation of the multilevel networks of Figure 31-1 is illustrated by the transfer of data from a lower network to the highest. The Navigation (NAV) subsystem, incorporating an intra-1553B bus network (Level 1), interfaces with the other two avionic subsystems and the Mission Computer via the Avionics Bus Network (Level 2), using contention protocol. At this higher level, processing can be accomplished on data functions common to all three avionic subsystems. At the third and highest level (Level 3) the Mission Computer interfaces with the remaining subsystems and common sensors via either the Global or Common Sensor buses using contention protocol. At this highest level data processing can take place that could involve any combination of subsystem functions. Note that the two flight critical subsystems (Flight Control and Propulsion) employ a common bus (Flight Critical), employing contention protocol as their means of integration. The Flight Control Processor provides the interface to the two higher order buses. There is a significant difference in the ITS configuration of the two subsystems. Wherein the Propulsion subsystem contains its own 1553B MUX bus network, the Flight Control subsystem employs the Flight Critical Bus as its intra subsystem network including the Propulsion Control Processor.

Such a topology provides functional isolation between the two flight critical subsystems and to all other subsystems and common sensors. A point of interest is that if a signal function originating in any of the three major avionic subsystems (NAV, COMM., FLT-INSTR.) is required in the propulsion subsystem, it must be processed through five (5) bus networks.

Figure 31-1 presents a hierarchical structure ITS with two HOX bus networks (Global and Common Sensor Buses) operating at the highest level, each with a specific assignment. The Global Bus used for interconnection of major subsystem and the Common Sensor Bus providing common sensor data to these same subsystems in a broadcast operating mode. These HOX networks operate with contention protocol and provide the functional isolation so important to flight critical subsystems.

Figure 31-2 presents another ITS configuration which incorporates only two levels of bus networks. It is comprised of two higher order bus networks, incorporating contention protocol, supporting their functionally related subsystem. Each subsystem (excluding the ICS-Command/Control) incorporates its own 1553B bus network. One HOX network, identified as the Flight Safety Bus provides data integration between those subsystems and sensor/units involved in basic flight of the aircraft. The Mission Bus provides the integration medium between mission/weapon type subsystems which have no direct function with basic flight.

Data/signal functions originating in the Mission Bus subsystems, required by the Flight Control and Propulsion systems, to assist in implementing mission/weapon tasks, are transferred through the Mission Management Processor (MMP). The MMP either processes and/or provides the storage for data transferred directly between the two HOX networks. The MMP broadcasts its data to the respective bus, like other subsystems on the buses, which it is serving. The Common Sensors have data paths to both HOX networks as do the Control/Display (C/D) subsystem(s) to provide maximum redundancy for both basic flight safety and mission functions.

Figure 31-3 illustrates the data flow through the hierarchical/multilevel bus network of Figure 31-2 that could occur with the detection of a hostile missile track in the Defensive Weapon Subsystem. Such detection, with appropriate aircraft/mission action, would require data transfer and processing at various ITS levels and involve many subsystems. The most critical would involve the transfer of data and command functions to the Flight Control and Propulsion subsystems to affect aircraft evasive action.

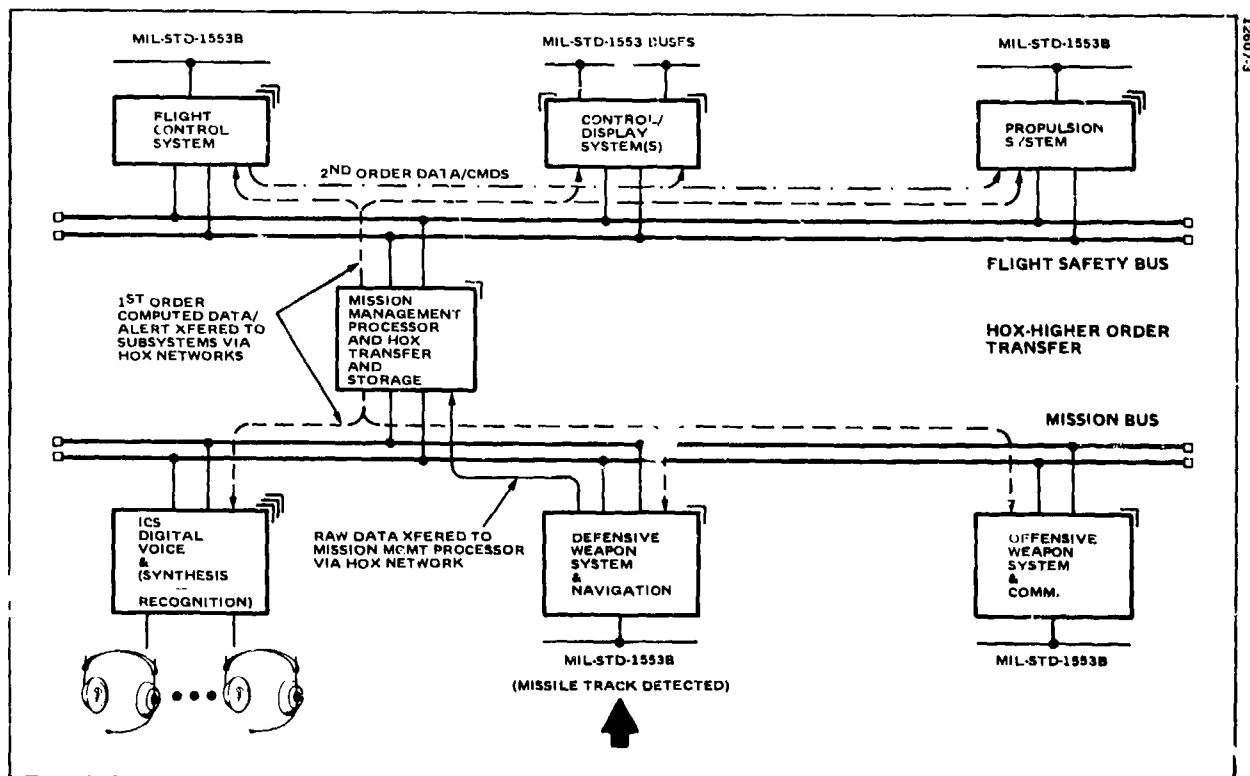


Figure 31-3. Illustration of Data Flow Between Hierarchical/Multilevel Bus Networks

Worthy of note is the postulated appearance of new subsystems and the functional combining of others. Such aircraft/system functions as Fire Control, Stores Management, AMAC, Weapon Guidance, and Release have been combined into one subsystem which contains its own 1553B bus network. A new subsystem identified as the ICS/Command Control subsystem was created to handle digital voice, audio control functions, and facilitate integration with the JTIDS digital audio channels. This subsystem will provide voice synthesis, and the all important voice recognition/command function, to improve the crew member's effectiveness by reducing his manual workload. This subsystem, unlike the majority of the others, would not incorporate a 1553B type ITS since its data transfer characteristics are not compatible with centralized command/response protocol but are with the contention protocol of a HOX system.

Two subsystems are identified as Control/Display (C/D) in Figure 3-2. These subsystems consolidate all those C/D functions that have previously been dedicated to the various avionic subsystems. Whether two are required depends on the size and mission tasks of the aircraft. The design concept and functional operational configuration of the C/D area is complicated because the cross-disciplinary engineering requirements are presently divided between many organizations, i.e., Human Factors, Flight Dynamics, Avionics, Propulsion, etc. One can foresee the need of single command/control selections via pushbuttons, e.g., the selection of a required attack mode in the Fire Control System which would result in an associated autopilot response, preparation of appropriate weapons, selection of a Radar System, selection of HUD symbology, selection of other displays, report via JTIDS, and etc. Further investigation will reveal whether C/D subsystems can be serviced by intra-1553B bus networks with special video switches and symbol generators or require a high speed ITS incorporating possible FM/FM operation on the bus or with the bus accepting processed/compressed digital video.

An imposing task even now confronting avionic system integration is the requirement that new aircraft integrate Flight Control and Fire Control subsystems to improve weapon delivery and gun laying accuracy. This integration task is complicated by the need of an integrated Interactive Propulsion/Flight Control System. One wonders how many cooks are going to be stirring this pot. The complications of the aforementioned task can be simplified by employing a higher order ITS to interconnect these critical avionic subsystems. Such an ITS incorporating contention type protocol, can preserve subsystem integrity, provide the avionics subsystem manufacturers independence in subsystem development, but most importantly, provide the means for integration.

Figure 31-4 is presented to show the physical and functional impact, a supposedly minor change made in the physical configuration of the bus network, can have on subsystem functional isolation, hardware minimization, dependence on state-of-the-art technology, etc. The connection of the two-HOX configuration (Figure 31-2) into a single HOX network (Figure 31-4) is certainly feasible, functionally. The single HOX network provides certain advantages over the dual configuration but also adds certain disadvantages. In-depth investigation would be required to determine if the advantages gained in the single network, primarily in elimination of hardware, would offset possible negative features of less isolation between flight and mission subsystems and less data path redundancy.

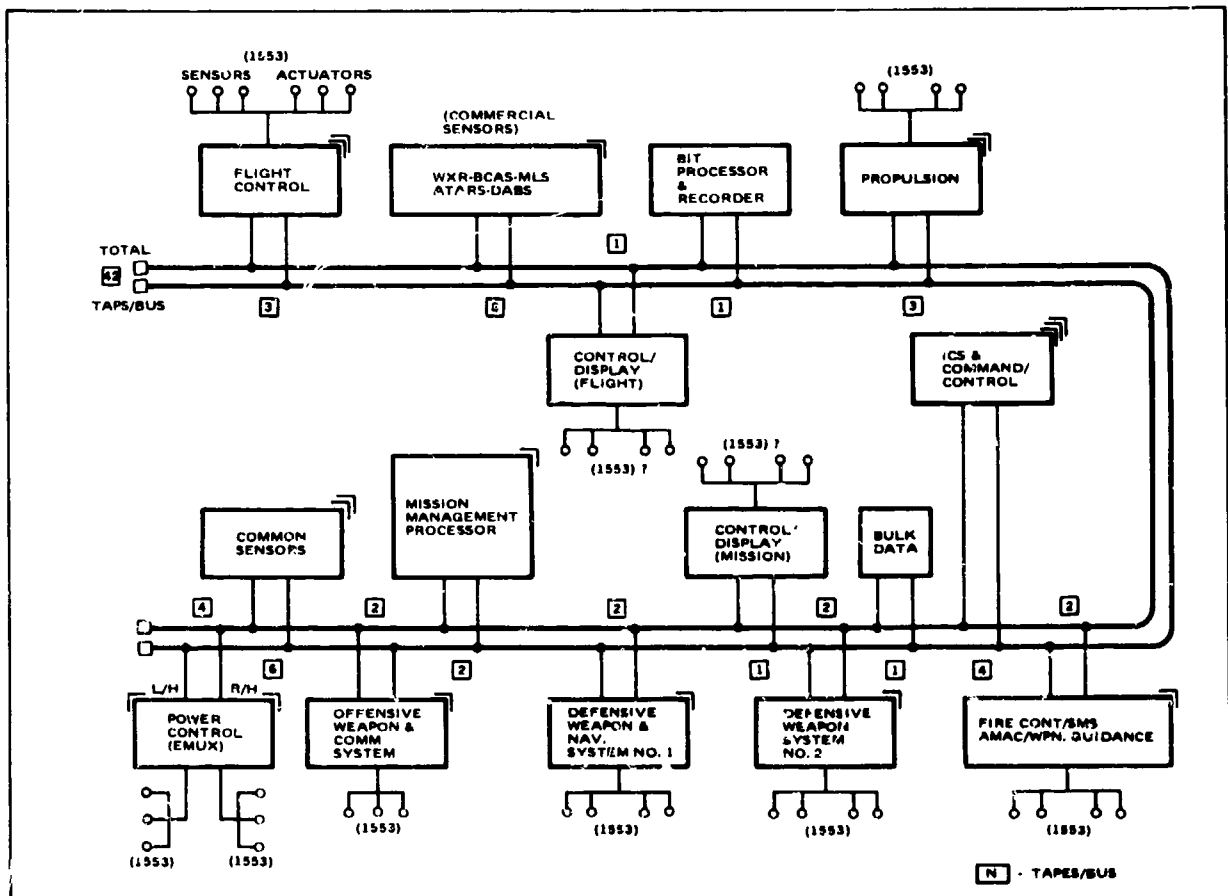


Figure 31-4. Two Level Information Transfer System (ITS) with Single-Higher Order Transfer (HOX) Network

A significant saving could be obtained, using the single HOX network by the elimination of hardware necessary to interface common subsystems and sensors to the dual HOX networks. Also, the Mission Management Processor (MMP) no longer need act as the transfer and storage agent for the interchange of data between the two HOX networks. The state of technology within the time frame of hardware implementation must be considered. The numbers shown in the squares at each bus-tap connection in Figures 31-2 and 31-4 gives the quantity of tap/drops required of each subsystem and sensor/unit per a bus network. In the dual HOX network configuration of Figure 31-2 a single bus in the Flight Safety Bus network has 26 tap/drops while a single bus in the mission bus network has 28. Fiber optics could not be used as the bus medium today. The single HOX network (Figure 31-4) has 42 taps/drops per bus. Could such a bus make use of fiber optics, not unless there is a major breakthrough.

AVIONIC SUBSYSTEMS/ITS CONFIGURATIONS

Each future avionics subsystem will have special data processing and transfer characteristics that will require certain intra- and inter-subsystem bus network configurations and protocol characteristics. Accepting the premise that each subsystem will contain its own multiplexed ITS and incorporate a data transfer interface to a HOX system, then each subsystem's physical and functional characteristics must be evaluated for data rates, accuracy, redundancy, isolation fault tolerance, etc., before the optimum intra- inter-ITS configuration can be determined.

Figure 31-5, an expanded version of Figure 31-2, illustrates various subsystem intra-bus networks and weapon system functions associated with each major subsystem. Figure 31-5 also presents a more in-depth picture illustrating the many different MIL-STD-1553B bus network configurations that can be adapted to the particular physical/functional requirements of the subsystem, remembering that there can be many network configurations developed, based on the ground rules and weighing factors selected. It is beyond the scope of this paper to present the reasoning for the choice of the different subsystems intra-inter-network configurations - although a Flight Control subsystem configuration is presented as an example of how subsystem characteristics and requirements can determine a selected ITS configuration and how the subsystem may use the HOX network to simplify its intra data-transfer task.

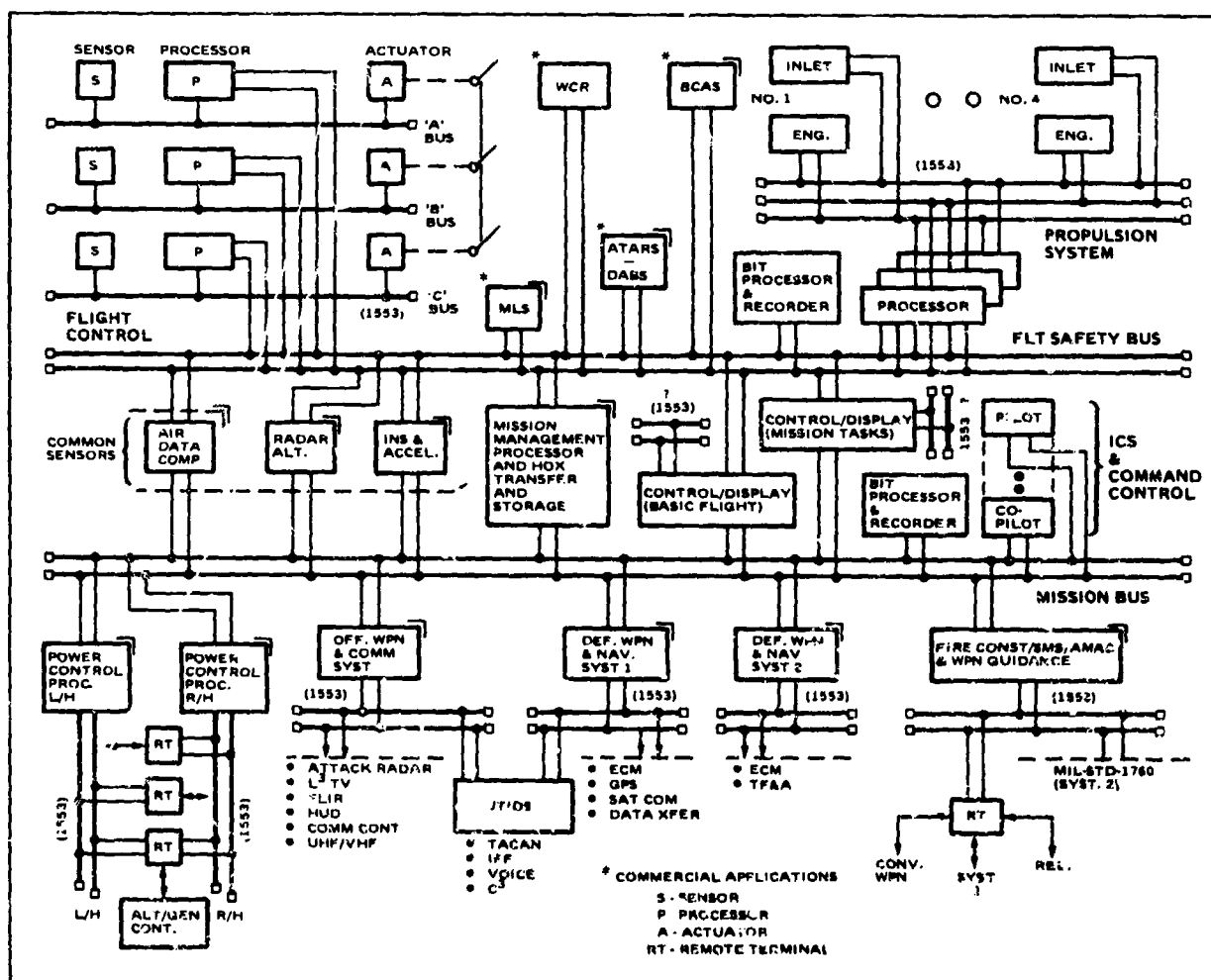


Figure 31-5. Two Level Information Transfer System (Showing Intra Subsystem ITS Configurations)

Figure 31-6 depicts a multiplexed, fly-by-wire, Flight Control System (FCS) - its intra/inter bus network configuration the result of tradeoffs that will be discussed. The FCS shown employs triple redundant 1553B data path/bus and interfaces with the two HOX networks, via the Flight Control Processors which integrates the FCS functions to the aircraft/mission functions. The FCS incorporates cross-strapping of triple redundant sensors/transducers to the three, single channel MIL-STD-1553B buses. Cross-strapping provides each Flight Control Processor (FCP) all inputs via its dedicated single channel Bus Controller (BC), bus, and Remote Terminals (RT). The cross strapping technique eliminates system hardware. For the FCP to receive all inputs from the triple redundant sensors, without cross strapping, the FCPs would require three BCs for connection to the three buses and each RT would also require three BIUs. This approach would probably meet the isolation/fault criteria required of a multiplexed fly-by-wire FCS, and it certainly reduces the hardware complement significantly, but one critical data path function is missing. Interchange of data between the FCPs, for voting and redundancy criteria, is not possible in the intra-bus network shown because it lacks intertie of the FCPs. This problem is solved free of charge by the HOX bus network which provides each FCP a redundant data path between FCPs - free of charge because the FCPs must have this connection to the HOX buses for aircraft/mission functions. Each FCP through its BIU can, by contention protocol, broadcast on the HOX buses that data required of the other two FCPs. This subsystem configuration is given as an example of how multiplexed bus networks, operating at different data rates and protocol, can be intertied in a manner that are complementary and the characteristics of each, even in a hierarchical/multilevel ITS, can provide functions not attainable in a single level bus network.

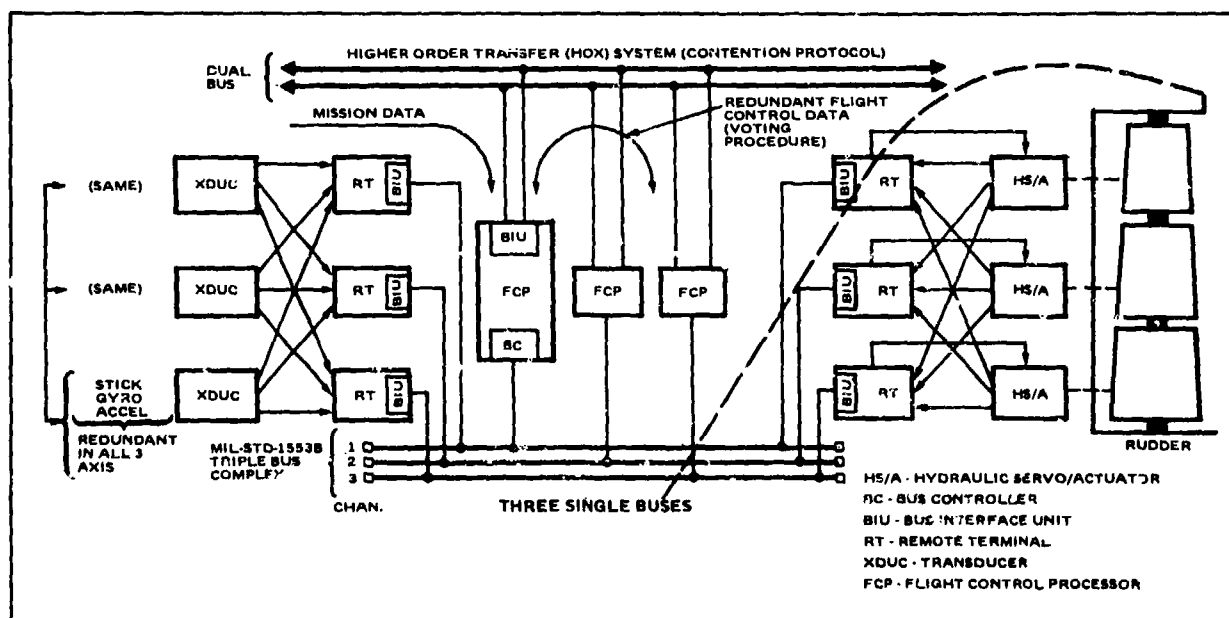


Figure 31-6. Multiplexed Fly-By-Wire Flight Control System

HOX SYSTEM MIL-STD-DEVELOPMENT

A Task Group (TG) formed within the SAE/A-2K Subcommittee on Multiplexing, has accepted the task of generating a MIL-STD for a Higher Order Transfer, Information Transfer System. The TG will need help, especially assistance and information from various agencies and organizations that are responsible for:

- Flight dynamic
- Avionic systems
- Propulsion/power generation and distribution
- Human factors and resources
- Standardization

Basic Information needed will include:

- Intra-and inter-subsystem data bases covering aircraft/mission subsystems configuration for various aircraft sizes and mission roles.
- Processing tasks/requirements of the Mission Management Processor(s).

With the above described information, the TG can conduct trade-off/analysis covering such HOX ITS areas as:

- Operational protocol-involving various addressing schemes, data flow paths, variable transfer speeds and message lengths, etc.
- Bus network and unit characteristics - involving bus length, number of bus tap/drops and bus/unit electrical characteristics.
- Bus network intertie and topology - involving means of providing the required level of unit/bus redundancy and fault isolation, plus the functional/physical intertie of hierarchical bus networks.
- Forecast of available technology.

It took ten years to develop MIL-STD-1553B. Its formulation was greatly assisted by the parallel development and use of various types of airborne multiplexed systems during the same period. The generation of a HOX ITS MIL-STD will be more difficult since there will be little development of hierarchical ITS within the period while generating the MIL-STD. A significant factor making the effort difficult will involve the ever changing functions of subsystems and the creation of new ones. The progress of the TG and the quality of the MIL-STD will depend upon the quality of information presented to it by the many military organizations whose task is to give direction and identify needs, and the industrial companies that manufacture the hardware/software elements respectively. Otherwise the MIL-STD will be a defacto standard generated by the TG.

First action of the SAE/A-2K, in developing the HOX ITS MIL-STD, was the conduction of an open-house meeting at WPAFB, Ohio, USA on 11 March 1980. Thirty-three (33) people attended the meeting and expressed their views on the MIL-STD development program. Each person attending was requested to submit what they believed to be the basic requirements of the HOX system including desired characteristics. Next meeting will be held in conjunction with the National Aerospace and Electronics Conference (NAECON) in May of this year.

CONCLUSIONS

A new Higher Order Transfer (HOX) type system (defined by a MIL-STD) is needed as the medium to interconnect avionic subsystems for total weapon system integration on next generation aircraft. MIL-STD-1553 data bus systems have given us a good start in the art of multiplexing and they should continue to be used to the maximum extent possible to provide subsystems with their intra-data transfer requirements. There is the need to tie all these independent operating subsystem bus networks together, and this can be accomplished by higher order systems operating between 1 and 50 MHz (possibly variable) with a contention type protocol. Effort is already underway by a Task Group sponsored by the SAE/A-2K Subcommittee on Multiplexing, to generate a MIL-STD covering such as ITS.

Information is requested by the TG covering new avionic subsystem functions/configuration requirements and associated data base lists. We must remember that the TG is comprised of volunteers and that its accomplishments will be determined by the support it receives.

GLOSSARY

AMAC	Aircraft Monitor and Control
ATARS	Automatic Traffic Advisory and Resolution System
BC	Bus Controller
BCAS	Beacon Collision Avoidance System
BIT	Built In Test
BIU	Bus Interface Unit
C/D	Control Display
DABS	Discrete Address Beacon System
DoD	Department of Defense
FCP	Flight Control Processor
FCS	Flight Control System
FM	Frequency Modulation
HOX	Higher Order Transfer
HUD	Heads Up Display
ICS	Intercommunication Subsystem
ITS	Information Transfer System
JTIDS	Joint Tactical Information Distribution System
MLS	Microwave Landing System
MMP	Mission Management Processor

RT	Remote Terminal
SAE	Society of Automotive Engineers
SMS	Stores Management System
TG	Task Group
WPAFB	Wright Patterson Air Force Base
WXR	Weather Radar

REFERENCES

- Bain, J. M., 1978, 'The Impact of Fiber Optic Multiplexing on Distributed Avionics Architecture,' 1978, Data Bus Conference, ASD-TR-78-34.
- Betts, R., 1980, '50 MBPS Fiber Optics Data Bus,' SAE/A-2K Presentation/Paper, IBM.
- Gross, J. P., Broadhead, S. L., Moore, J. D., 1980, 'IMUX: High Speed Communication Bus,' SAE/A-2K, Presentation/Paper, S.C.I., Inc/University of Alabama.
- Husbands, C. R., 1979, 'Airborne Integrated Communication System,' 3rd Digital Avionics Systems Conference.
- Smith, L. A., Crossgrove, W. A., Dervey, D. E., 'Advanced Avionic Systems for Multimission Applications,' AFAWL Report F33615-77-C-1252, Boeing Military Airplane Company.
- Swaney, R. E., 1980, 'C³I Data Bus,' IR&D Report, Hughes Aircraft Company.
- Whiting, J. H., 1979, 'Military Aircraft Avionics in the 1980's,' Standardization in Military Avionics System Architecture Symposium, WPAFB, Ohio.
- Metcalf, P. M., Boggs, D. R., 'Ethernet: Distributed Packet Switching for Local Computer Networks,' Communications of the ACM, July 1976.

DISCUSSIONS SESSION VI

REFERENCE NO. OF PAPER: VI-21

DISCUSSOR'S NAME: Erwin Bengt, WPAFB, USA

AUTHOR'S NAME: I. Moir (P. Duke, presenter)

COMMENT: You seem to imply that MIL-STD-1760 does not totally satisfy your requirements and is too complex. Since this standard is in the final coordination cycle do you have any additional inputs? Concerns?

AUTHOR'S REPLY: British Aerospace-Brough had a number of comments to make on MIL-STD-1760. Our comments were sent to Smiths Industries who were tasked with collating a UK industry response. This response was duly sent to the US but British Aerospace-Brough have had no further information on the state of 1760. In view of the serious implications of some of our comments, I would be very interested to see a revised version of this standard, hopefully before it is "frozen."

REFERENCE NO. OF PAPER: VI-28

DISCUSSOR'S NAME: CDR Strada, USN

AUTHOR'S NAME: Moir (P. Duke)

COMMENT: How do you intend to accomplish "tuning the system to pilot capabilities"? How would you handle the calibration constants/initial conditions that the weapon-aiming system needs to know about the weapon? These constants may vary from aircraft to aircraft as well as weapon to weapon.

AUTHOR'S REPLY: (1) The pilot's capability to interact with the aircraft system was seen from the outset to be very important. To give a brief history, the Rig has grown from two activities at British Aerospace Brough. The first was research into data bus systems from an avionic point of view and the second was the development of an advanced cockpit for a single-seat tactical combat aircraft. The latter has been used to perform ergonomic assessments of new control and display concepts. Extensive "outside world" and data analysis features have been added to provide a complete tool for the assessment of future systems. "Tuning" the system will be achieved by modifying the cockpit and avionic systems. The avionic systems will initially be simulations, building up through emulations to "real" hardware and software. Hence, modification during the early development of the system will be relatively cheap.

(2) Navigation parameters will probably be input to the aircraft using a portable on-board data source. The extension of this to weapon data is controversial and requires study. In the absence of a method for modifying the constants, I would suggest more use of the role change philosophy. Certain LRUs which contain preset data should be easily replaced. In general, the rule is that the most feasible system will contain the minimum hardware and software dedicated to a specific weapon type.

REFERENCE NO. OF PAPER: VI-28

DISCUSSOR'S NAME: J. F. Ferreri, DASSAULT

AUTHOR'S NAME: Moir (P. A. Duke)

COMMENT: Comment comptez vous résoudre le problème des interfaces analogiques et en particulier les signaux discrets de tri compte tenu de la standardisation que vous souhaitez obtenir.

How do you intend to solve the problem of analog interfaces and in particular the use of discrete signals taking account of the standardization you are looking for?

AUTHOR'S REPLY: This depends on the aircraft/launcher/weapon interface. However, for conventional weapons the analogue and discrete signals would be generated within a Pylon Interface Unit by D to A conversion or switching of power supplies. Only power supplies and digital data would be input to the Pylon Interface Unit.

REFERENCE NO. OF PAPER: VI-29

DISCUSSOR'S NAME: Schoelch, IABG

AUTHOR'S NAME: HEGER

COMMENT: How do you achieve tolerance against interruptions of the fiber optic bus?

AUTHOR'S REPLY: In normal ring operation one transmission direction is used (out of two possible). In this operation mode several messages can be conveyed simultaneously using the principle DDD, the messages run from the source station till the destination station where are absorbed. On-line segments

where no message transmission takes place special delimiter symbols are transmitted. In case of live interruption (broken transmission medium in both directions or faulty station) this is detected by the adjacent station by means of a time-out. This station initializes the reconfiguration procedure; this leads to two stations which recognize themselves as being adjacent to the interruption. These two stations reverse the transmission direction alternatively and periodically by means of special broadcast messages. Messages having certain destination addresses are only transmitted within the respective period with the adequate transmission direction. By reversing procedure no messages are lost but the message flow is only choked and so the average throughput is not affected in the main, but the average transmission times become longer. When an additional interruption happens, the same procedure as described is performed. When a station at the end of the physical line receives bits from the so far interrupted line (again) the new or repaired part of the line with its stations is identified and coupled (again). And finally, when all interruptions of the ring bus are closed the system reinstalls the ring structure with an arbitrary transmission direction.

In the case of the interruption of only one of the two transmission directions the other direction is selected and full performance is guaranteed.

Besides interruptions direct effects are also detected, and the respective line reconfiguration takes place as well.

And finally it must be pointed out that all changes of the configurations are reported system-wide by means of broadcast status reporting messages and so the line and system status is displayed on the display of the master control panel in order to be able to inform the repair personnel effectively.

REFERENCE NO. OF PAPER: VI-30

DISCUSSOR'S NAME: G. H. Hunt, RAE

AUTHOR'S NAME: Megna

COMMENT: In your paper you mention the objective of comparing the dispersed sensor mesh system with the existing dedicated system already developed for the F-8 fly-by-wire aircraft. Could you state whether this comparison has yet been made and give an indication of the results obtained.

AUTHOR'S REPLY: An F-8 iron bird facility was used which has the flight system as a part of it--the network is in parallel with that. The comparison is made based upon examination of the functions performed by the two different implementations.

REFERENCE NO. OF PAPER: VI-30

DISCUSSOR'S NAME: Alan Stern, Boeing Co., USA

AUTHOR'S NAME: V. Megna

COMMENT: You seem to be attempting to solve the problem of interfacing sensors and actuators to flight control system computers using a flight safety reliable bus. Why isn't a redundant 1553B approach good enough for this? What is the advantage of your approach relative to 1553B?

AUTHOR'S REPLY: There are a number of bus problems which we hope to avoid through the use of a network. One is physical damage which results in loss of communication to units beyond the break, another is the problem of some subsystem disabling the bus by constantly transmitting on the bus. But, basically we are investigating the use of a network concept to establish a data base upon which to make decisions as to the best method for interconnecting avionic subsystems.

REFERENCE NO. OF PAPER: VI-30

DISCUSSOR'S NAME: Horst Kister, VDO

AUTHOR'S NAME: V. Megna

COMMENT: The controller is the most critical part of the system in respect to safety. If each of the terminals were able to disconnect the bus from itself without any handover mechanism (not like dynamic bus control, but "automatically"), would that help?

AUTHOR'S REPLY: Distributing bus control complicates the problem. Central bus control makes for less complexity, but requires extra reliability consideration. Complexity comes about when control is passed.

REFERENCE NO. OF PAPER: VI-30

DISCUSSOR'S NAME: Horst Kister, VDO

AUTHOR'S NAME: V. Megna

COMMENT: The controller is the most critical part of the system in respect to safety. If each of the terminals were able to disconnect the bus from itself without any handover mechanism (not like dynamic bus control, but "automatically"), would that help?

AUTHOR'S REPLY: Distributing bus control complicates the problem. Central bus control makes for less complexity, but requires extra reliability consideration. Complexity comes about when control is passed.

REFERENCE NO. OF PAPER: VI 30

DISCUSSOR'S NAME: K. Brammer, ESG

AUTHOR'S NAME: V. Megna

COMMENT: The communication network shown by you has 6 nodes. If they were all mutually connected to each other, every node would have 5 ports to the other nodes and there would be a total number of 15 links (i.e., $N(n-1)/2$ links with $N=6$). You did not choose this maximum configuration, but some configuration between the maximum and the minimum possible. Is there a particular reason for the selection of your configuration? Has it to do with a specified degree of redundancy (e.g., for flight safety) or did you design for a specified cut set (the minimum number of link failures that cause the net to split into two separate parts)?

AUTHOR'S REPLY: Every node has 4 ports and the controller has 4 ports. With this configuration you do not end up with unconnected ports--all will be used. Otherwise, you end up with a port that cannot be connected to anything else.

REFERENCE NO. OF PAPER: VI-30

DISCUSSOR'S NAME: E. Gangl, WPAFB, USA

AUTHOR'S NAME: Megna

COMMENT: You mentioned in your discussion that MIL-STD-1553 LSI hardware was not available and also that there was no standardization guidance on fiber optic bussing. I would like to mention that the US is considering a fiber optic version of 1553 (probably will be MIL-STD-1773) for publication. Also that LSI 1553 terminal hardware will be available this year from several sources. In the US from Harris Corp., Collins, Grumman, Circuit Technology, Inc., and in the UK from Smiths and Marconi Electronic Devices.

AUTHOR'S REPLY: As I mentioned in my presentation, at the time that we were designing and constructing our system MIL-STD-1553 LSI chips were not available and therefore, they were not included. This does not preclude the addition of 1553 LSI chips when they are available. As far as a specification for 1553 fiber optics is concerned, the system which we have built is an engineering model to test out our network concepts. If we do go on to build a flight system, we will take in consideration any fiber optic standard which exists at that time.

REFERENCE NO. OF PAPER: VI-31

DISCUSSOR'S NAME: Alan Stern, Boeing Co., USA

AUTHOR'S NAME: J. McGuen

COMMENT: Because it is desirable to reduce the number of buses and interfaces to a minimum, and because MIL STD 1553 is strongly encouraged even within flight control systems; it is desirable to provide a 1553 mode which is a "contention scheme." This would prevent the need for additional bus redundancy management in flight safety critical systems.

AUTHOR'S REPLY: Identify a contention Information Transfer System (ITS) as one wherein each Remote Terminal (RT) has the means of acquiring the bus under its own control. There is no central control needed or allowed as in a 1553 system.

Since a 1553 system has central control, even operating in a "dynamic bus control" it cannot ever operate as a pure contention system. Even a 1553 system operating in a dynamic control mode must keep handing off control from RT to RT in a set sequence. The U.S. tri-services have determined there will be no change in 1553B, i.e., no 1553C allowed.

Also, a 1553 bus cannot provide the functional isolation from other subsystem (RT) failures as can a contention bus system. Bus management in a contention-type system would be minimal.

SIFT - AN ULTRA-RELIABLE AVIONIC COMPUTING SYSTEM

Kurt Moses
Bendix Corporation
Flight Systems Division
Teterboro, New Jersey, USA

SUMMARY

SIFT (Software Implemented Fault Tolerance), is an ultra-reliable computing system that is designed for flight-critical control and avionics applications. A typical application would be a fly-by-wire control system for civil or military aircraft. SIFT is based on a multi-processor architecture that achieves fault tolerance by replicating computing tasks among processing units. Error detection and system configuration are performed by software to maintain the operational integrity of the computing system. SIFT has been designed to meet a system failure probability goal of 10^{-10} per hour.

SIFT operation requires a high speed inter-computer communication system. This is realized by dedicated serial links arrayed in a star connection, i.e. every processor broadcasts to and receives data from all other processors in the complex. Care has been taken that no delay due to contention for ports, buses and processors, limits system operation.

Computing is carried out by high speed, 16 bit Bendix 930 processors, which have a throughput of approximately 800 KOPS based on an appropriate flight control instruction mix. Each processor has a 32K memory associated with it.

Software algorithms are used for failure detection by means of voting, failure isolation to the faulty processor, and reconfiguration after fault detection. Frame synchronization between processors is employed to reduce data skew and minimize false alarms.

This paper describes the architecture of SIFT, its hardware implementation, and the unique test stand used for evaluation. Potential applications of this technique to current and anticipated ultra-reliable electrical flight control systems are given.

The work presented in this paper was done by Bendix Flight Systems Division for SRI International under NASA contract number NAS1-15428. This work is being sponsored by NASA Langley Research Center.

1. INTRODUCTION

Automatic flight control systems which once provided mainly pilot-relief functions, have in recent years taken on flight-critical tasks, i.e. tasks whose successful accomplishment is vital to the safety of the aircraft. Automatic landing under low visibility/ceiling conditions was one of the first of these flight-critical tasks to be imposed on the AFCS. More recently, "fly-by-wire" (electrical) control systems have taken the place of conventional mechanical controls, and the Control-Configured Vehicle (CCV) which achieves the desired flying characteristics at least partly by means of electronic controls, rather than solely by aerodynamic configuration, has made its debut. The desire to reduce fuel consumption has given a powerful impetus to the use of electrical flight controls, since this permits the unaugmented aircraft to be designed in a minimum drag configuration. The vehicle then achieves satisfactory flying qualities through the use of the electrical flight control system. Mechanical controls then become superfluous in such a vehicle since, without the electrical system, the vehicle is uncontrollable (un-flyable). Obviously, a flight control system entrusted with such tasks must be ultra-reliable, i.e. its reliability must be of the order of the basic aircraft structure. These considerations have led to the development of SIFT, which achieves the failure probability of 10^{-10} per hour through the use of software-implemented fault tolerance techniques and hardware redundancy.

While primarily motivated by the requirements of flight control and related flight-critical applications (e.g. flutter control, engine fuel control etc.), SIFT can be used in the context of the total avionic system for both flight-critical and non-critical tasks to achieve an overall avionic system that may be more economical than the present accumulation of separately designed LRU's. These often cannot even communicate with each other, much less substitute for one another. With SIFT, it is possible to substitute a failed processor that was performing a critical task with one that is performing a less critical task, and processor inter-communications are handled in a routine manner. SIFT is a multi failure-survivable, multi-processor computer array that utilizes dedicated ports and busses for all interprocessor data transmissions so that there are no major delays due to contention. All fault detection and reconfiguration algorithms are implemented in software.

Each processor communicates with the other processors over bit-serial busses by broadcasting its computed data. This data is validated by means of, 3 or 5 fold voting, with presumably identical data broadcast by the other processors. Voting is done exclusively by software. Majority voting is most effective if the values subjected to vote are identical except for errors. The computed results can only be expected to be identical if the programs receive identical inputs. This in turn requires some degree of synchronization between processors and further, requires a basic strategy to insure input data

consistency. These requirements impose a large interprocessor communication load on the bus system which could lead to unacceptable delays due to contention for bus or data port access in multiplexed busses.

External I/O information is transferred by MIL-STD-1553A serial links. Time division multiplex controllers govern the data flow to and from aircraft actuators and sensors. There is one controller for each processor and 1553A bus. Each 1553A controller and bus can support up to 32 remote terminals with associated actuators or sensors.

The SIFT hardware design, build, and test effort was the responsibility of Bendix Flight Systems Division, under contract to SRI, International who is the prime contractor to NASA Langley Research Center under NASA Contract NAS1-15428.

2. SYSTEM DESCRIPTION

The present system has been designed to accommodate up to 8 processors, a Software Development System, fault tolerant redundant power supplies and 8 1553A terminals that can connect the SIFT processors to sensors, actuators, controls and displays. Each processor is capable of executing a complete control program, typically including fly-by-wire control, stability and control augmentation, autopilot modes including autoland functions, navigation, guidance, etc. Each processor also executes the redundancy management algorithms including fault diagnosis and reconfiguration strategies, as well as the executive program. Although not included in the SIFT development program, a preflight and maintenance BIT program will be required for most operational avionic applications of SIFT.

Typically, a flight-control application program includes the processing of sensor data and control inputs (filtering and otherwise shaping the data data, voting and comparing of redundant data); the generation, by means of the applicable control laws, of actuator and instrumentation commands and other outputs; the engage, disengage and mode control logic; fault and other types of warning displays; and ensuring the integrity of the output commands by appropriate monitoring and switching logic. In addition, and as a characteristic of SIFT, all computed data that is transmitted between processors is subjected to the software voting algorithms, and failures in any processor are communicated to all processors.

Figure 1 illustrates the arrangement of the SIFT architecture and shows the interprocessor serial bus structure and the I/O data link which communicates with the other constituents of the aircraft flight control system. The I/O data link is a MIL-STD-1553A bus. Each bus can communicate with 32 remote terminals.

3. SYSTEM OPERATION

The organization of each computer-LRU is shown in Figure 2. The CPU is a Bendix 930 minicomputer. Computations and broadcasts of data are carried out in an iterative sequence. The result of the computations are temporarily stored in the scratch pad memory data file (1K) that is uniquely associated with the processor. Each processor has associated with it its own program memory. This memory may be read by but cannot be written into by any other processor. The data file can be accessed by the broadcast transmitter, the receiver, the 1553A data link and the CPU in this order of priority. This sequence has three phases which control the activities of the system components.

Load Phase. The processor computes its assigned tasks, loads resultant data into its local "data file", loads the associated destination address into its transaction file, and loads the starting transaction address into the transaction pointer. The broadcast sequencer then starts the Broadcast Phase of operation followed by the Receiver Phase in each destination processor. It should be noted here that the Broadcast and Receiver phases described below function independently of the processors and do not detract from the power and speed of the CPU's that make up the SIFT Computer System.

Broadcast Phase. The broadcast sequencer broadcasts a data word (from "data file") along with the associated destination address (from "transaction file") at a maximum rate of 1 data word/15 microseconds. This broadcast sequence continues until End-of-File (EOF) is reached in the "transaction file." The flow diagram for this sequence of events is shown in Figure 3. End-of-File (EOF) is reset by loading the transaction pointer with the starting address. The 16-bit data file word is then combined with the 7-bit destination address in the broadcast transmitter (Figure 4). The 25-bit serial word is then concurrently broadcast to all other processors in the system. The EOF is updated, and the transaction pointer is advanced to the next transaction if additional data words are required by the program. Otherwise, the sequence of broadcasts is terminated.

Receiver Phase. The bit-serial word is transmitted in synchronism with a 4 MHz clock over busses that are dedicated to each destination processor. The transmitted word is stored momentarily in dedicated receivers in the destination processors. Here, receiver sequencers (Figure 5) scan the receivers for full registers, then steer the data words to the local data file locations indicated by the destination addresses. All receiving processors receive the same data words and store these data words at the same relative locations in their local data file. The maximum time to load a received word into the data file is 9.12 microseconds, the minimum time is less than 1 microsecond.

4. CPU DESCRIPTION

Central Processor. The CPU selected for the SIFT is the BDX-930, the latest in a line of Bendix series 900 processors. The BDX-930 is a 16-bit, microprogrammed, parallel, general-purpose machine employing a 2901 bit-slice ALU (Arithmetic Logic Unit). The architecture, integrated by Bendix with the latest standard "off-the-shelf" MSI and LSI components, results in a processor specifically tailored for high-speed real-time flight control computations qualified for military applications.

The BDX-930 CPU is constructed using a family of bipolar micro-processor devices supported with low power Schottky MSI arrays, thereby providing maximum computational capability in a minimum power and size configuration.

The computer performs 16-bit parallel arithmetic operations during its micro-command execution time. To maximize execution speed, an instruction-stream pipeline organization is used which provides concurrent fetch, decode, and execute operations, together with a pipelined microprogrammed sequencer and broad micro-control field. Therefore, many simultaneous functions can be performed at maximum speed.

In addition, there are separate memory address and data buses to increase the throughput with the memory. To interface with the slower operating speeds of core memories and various I/O devices, a request/response system is used to lengthen those micro-orders in which communications with these external elements is necessary. The BDX-930 contains 21 registers which are usable by the programmer. Sixteen of these serve as general purpose accumulators, while the remaining six include the program counter, switch register, and four specialized single bit registers.

Accumulators 0 through 15 are used as general purpose accumulators, providing the capability for most machine operations. The registers are operated upon primarily through use of a powerful set of inter-register instructions. Provision is also made to utilize two of the registers as index registers during memory reference operations, and one register as a stack pointer in stack related operations. In addition, sequential registers are automatically linked for double precision operations.

The use of high-performance Schottky transistor-transistor logic elements permits extremely fast internal clocking rates - as high as 16 megahertz (62.5 nsec period). This produces a CPU cycle time of 250 nanoseconds and an average operations rate of 942 KOPS. Inter-register ADD is executed in 250 nanoseconds; firmware-based MULTIPLY is executed in 5.1 microseconds.

The BDX 930 consists of 86 microcircuits mounted on one printed circuit board (approximately 50 in.²).

5. MEMORY

Memory addresses are logically subdivided into mapped segments as shown in Figure 6.

Each processor's main memory and stack contain 30K words, each word 16 bits long. This memory holds the SIFT executive program, the application program, and the control stack. As noted, the significant results of each processor's computations are temporarily stored in a scratch pad memory data file. Each data file contains 1K data words, each word 16 bits long.

High speed interprocessor communication is provided by separate processor/bus interface elements which control the bit-serial transmission and reception of data words. The memory destination of each transmission is provided by the transaction file in each processor. Each transaction file contains 1K words, each word 16 bits long.

Discrete Functions. A reserved block of 8 addresses is used to address 12 discrete functions that are firmware or hardware implemented (see Figure 7). These functions increase the power and speed of the SIFT Computer System. The implemented functions include:

- read processor identity number
- set EOF
- read real-time-clock
- write (set) real-time-clock
- read 1553A registers
- write 1553A registers

External I/O. External I/O information is transferred by MIL-STD-1553A serial links. Time division multiplex controllers govern the data flow between aircraft actuators, sensors, avionics modules, and the BDX 930 processor. There is one controller for each BDX 930 processor and 1553A bus. Each 1553A controller and bus can support up to 32 remote terminals with associated actuators, sensors, or avionics modules.

The 1553A controller is a 32-bit-word-sized, microcoded processor. It has address computation capability, microcoded test routines, ability to program branch and special purpose registers and ability to operate on a prioritized interrupt or polling basis.

The controller shares memory with one BDX 930 processor through the data file of the processor. The interface to the data file is parallel by 16-bit word. The interface to the 1553A bus is serial by bit. The 1553A controller consists of two major sections:

The analog section is a waveform and impedance converter. It converts the pulsed digital data received from the digital section to a 1 megabit serial 1553A bus-compatible signal for transmission over the 1553A bus. In turn, it converts the received 1553A bus signals to pulsed digital data that can be processed by the digital section. The digital section responds to commands from the BDX 930 processor to transmit, receive, or idle. In addition, it encodes and decodes bus data as required by mode logic. The digital section is firmware-programmable with respect to parity sense, host processor byte/word requirements, inter-word gap length and error routines. The present microcoded routines handle these error situations:

- early RT response
- late or no RT response
- incorrect response word
- inter-word gap too long
- invalid sync or parity
- Manchester errors

The 1553A controller consists of 64 microcircuits and a miniature transformer mounted on one printed circuit board.

Computer Specifications. One SIFT computer/Processor LRU is composed of 9 modules. Computer functions are allocated to these modules as follows:

MODULE	FUNCTION
BDX 930	CPU
Memory #1, #2	Main Memory, 15K words each
Timing & Control	Timing for CPU Real-Time Clock I/O Bus Interface Logic Control Logic
Memory Interface & Control	Transaction File and Logic Data File and Logic Transaction Pointer Timing for Broadcast (4 MHz) Broadcast Shift Register
Processor Interface	Broadcast Sequencer Receiver Sequencer
Broadcast/Receiver	Broadcast Drivers Receiver Shift Registers Holding Registers
1553A Controller	1553A Controller Functions with BDX 930 Interface
Power Supply	+5V D.C. 13 amps +15V D.C. 1 amp

Physical Characteristics. Each SIFT LRU conforms to ARINC 404A packaging and is a $\frac{1}{2}$ ATR short unit. It weighs approximately 13 lbs. and has a computed MTBF of 8,900 hours. Cooling is self-contained by means of an internally mounted fan. Construction is per conventional multi-layered printed circuit boards. 25% growth space is provided in each unit. Power consumption is 90 watts.

Software Development System. Software generated for the SIFT system can be developed on a Data General Eclipse minicomputer system. The equipment is capable of:

- compiling, assembling, linking and testing the SIFT code
- loading the digital processors
- providing real-time monitoring, timing and alteration of executing programs
- providing automated documentation and change control
- providing automated module testing
- providing control law evaluation capability under a simulated airplane/flight profile environment

The equipment comprising the Software Development System consists of:

- A Data General Eclipse minicomputer chassis containing an Eclipse S/230 mini-computer with 128K bytes of core memory, a floating-point unit, and disk, console, line printer, and flight-processor computer interface modules.
- a 10-megabyte disk unit
- a console
- a line printer
- interface modules
- an EIA interface module

All operations are performed under software control executed on an Eclipse S/230 Data General computer. A maximum of eight Bendix BDX 930 computers can be controlled with the current SDS interface.

The interface between the Eclipse S/230 and the Bendix BDX 930 consists of two modules. The first module is a standard 15" x 15" card that plugs into any unused I/O slot of the S/230.

The second module is mounted in a rack mounted chassis that connects with the first module through a standard D3 4192 paddleboard connector wired to the unused I/O slot of the S/230 computer backplane. The Bendix BDX 930 computers are connected to the rear of the second module through their respective access panel cables. The Eclipse S/230 governs all transfer of data to and from the SDS by means of I/O instructions. Resident in the Eclipse S/230 is the SDS software; software to control a Data General Nova 3/12 Computer which controls the 1553A terminals; and the software to simulate sensors and actuators, as well as airframe characteristics suitable to a particular application to demonstrate the capability to actively control a simulated airplane. Figure 8 shows the overall system configuration (including the test set-up). The vertical dotted line indicates the hardware resident in the two electronic bays shown in Figure 8. One controller controls the 1553A communication link, one DMA controller is needed for the interprocessor bus transmitter, and the third one for the bus receiver as shown in Figure 8.

Testing. In order to validate the design to the confidence level required by the specification, extensive tests were planned. These include hardware tests, system tests, and software validation tests. Hardware tests ensure the correctness of the design implementation, specifically timings, interface circuitry operations, and integrity of construction. System tests are conducted both on developmental models of SIFT and on any later production versions. These tests will exercise SIFT in both open and closed loop modes. Loop closure is achieved by tie-in to a simulation of aircraft dynamics and sensors on a general purpose minicomputer (e.g., Data General Eclipse, PDP-11, etc.).

A general block diagram of such an arrangement is shown in Figure 8. The purpose of system tests is to ensure dynamic stability, achievement of static and dynamic system accuracy requirements, validation of execution time estimates and inter-sample ripple characteristics, and observing system behavior in the presence of injected faults. These and similar characteristics of the system can only be evaluated in a dynamic environment that is as similar to the aircraft environment as possible. In view of their cost and the difficulties of controlling flight test conditions, flight tests are only used for final system validation and certification, and very rarely for development purposes. Because SIFT depends on correct software, not only for the application program, but specifically, for voting, fault detection and isolation, and reconfiguration, it is essential that the software be validated to an extremely high level of confidence. Software quality will be enhanced by the use of a structured language for the higher level software. Extensive software tests are planned on the prototype SIFT system and these tests have been proceeding at SRI, International. These tests will encompass complete flight conditions, environmental conditions, single and multiple failures, crew inputs, etc. Later on, much of the testing will also be performed at NASA Langley Research Center.

The SIFT hardware has undergone extensive testing, and an effective 'on line' time of 3 months was achieved with less than 1% infant mortality rate experienced during the initial first week of testing per box. Figure 9 shows the hardware configuration.

Prior to delivery to SRI, the following tests were conducted.

- A. The extended Bendix BDX 930 CPU Test was conducted on all 7 SIFT computers and run continuously for 30 minutes each with no failures. This tests all the instructions in the BDX 930 repertoire. A memory test was conducted on all the SIFT computers in which bit patterns were written and then verified in all memory locations, main memory, transaction file, data file, and discrete registers. This test was conducted continuously for 20 minutes each with no failures.
- B. Interprocessor data transfer tests were conducted where each processor's ID was stored in the upper data file location starting at location 7780₁₆ to 77F7₁₆. All SIFT processors then broadcast simultaneously their respective ID values to all other processors. Each data file was printed out and verified as to correct data content. Each SIFT Processor was moved to a new ID location and the test was repeated. An additional test was also conducted where the Real-Time-Clock value of each processor was transmitted once to each other processor and then verified by reading the data file. This test was conducted for each data file location. These tests were conducted at full clock rates and no failures were noted.
- C. All SDS (Software Development System) functions were performed simultaneously on the 7 SIFT processors including halt, read, load, and restart with no failures noted.
- D. Data transfer tests over the 1553A Data Link were conducted for blocks of words from 1 50 32 in Controller to Terminal and Terminal to Controller mode for all SIFT processors simultaneously. Terminal to Terminal transmission mode tests were not conducted since the system is not configured for this mode of operation. This capability is provided for in each processor.

All the tests were conducted at the nominal input voltage of 28 volts DC, and again at 32 volts DC and 24 volts DC, with no failures noted.

Concluding Remarks. A brief description of the SIFT system, with emphasis on implementation, has been presented. The system, a multi-processor computing system that relies on software - implemented fault detection and reconfiguration algorithms, is an efficient approach to the design of ultra-reliable avionics. Its development will pave the way for the acceptance of fly-by-wire and other advanced flight control systems.

Potential applications of SIFT include all new technology aircraft, including the energy-efficient transport, military developments such as a new strategic bomber, and spacecraft such as an Advanced Shuttle. SIFT should be considered whenever flight control system survivability requirements can not be satisfied with conventional triplex or dual-dual configurations. It should also be considered in the context of integrated avionic systems having a mixture of flight-critical and non-flight critical sub-systems.

SIFT ARCHITECTURE

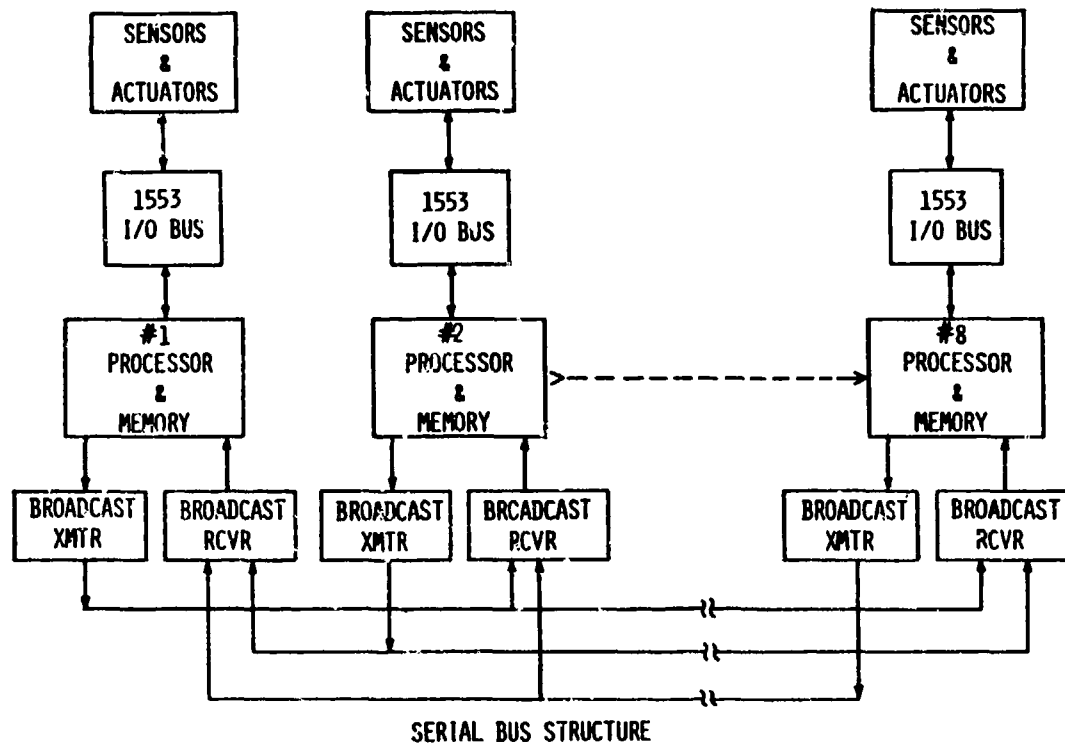


FIGURE 1

SIFT COMPUTER ORGANIZATION

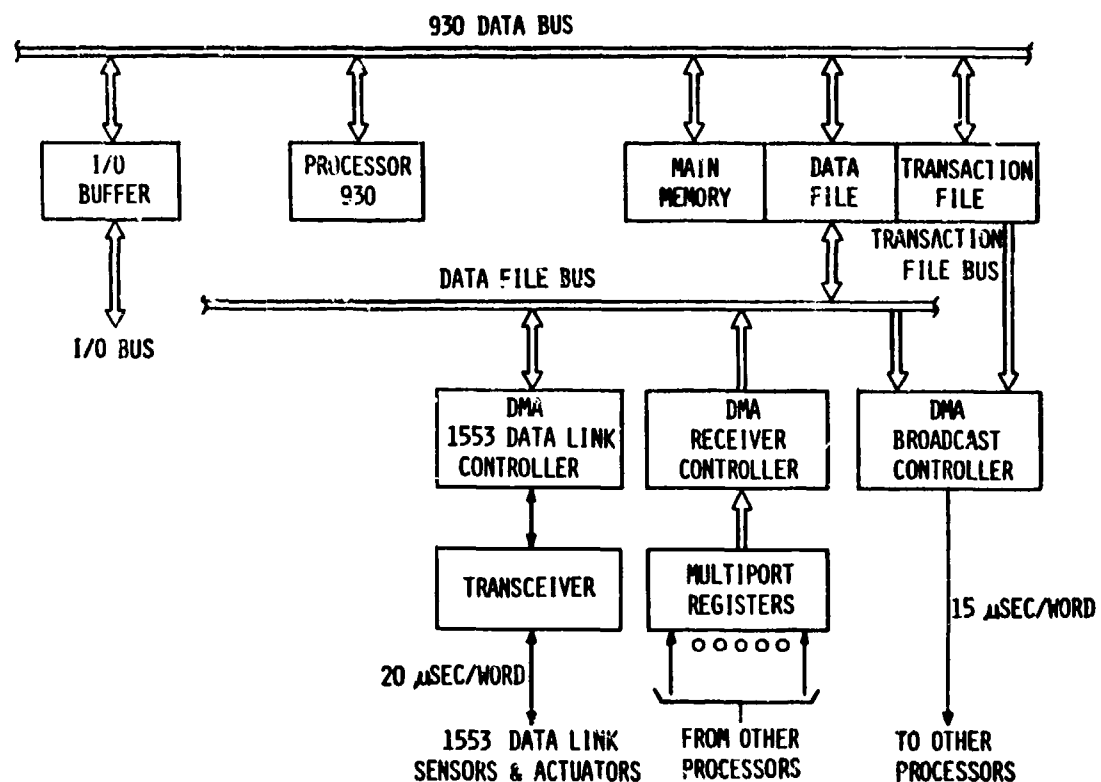


FIGURE 2

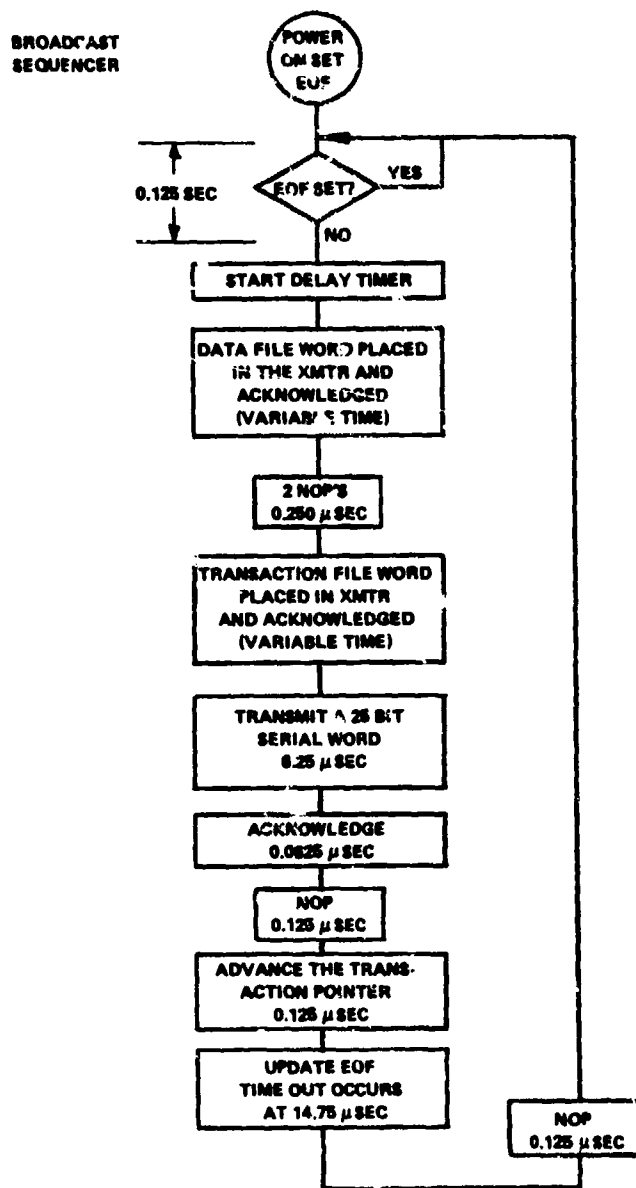
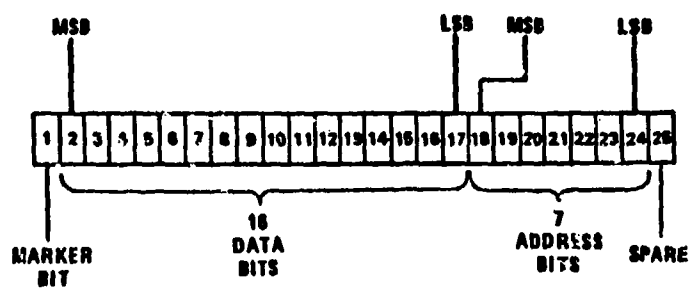


FIGURE 3



SERIAL WORD FORMAY

FIGURE 4

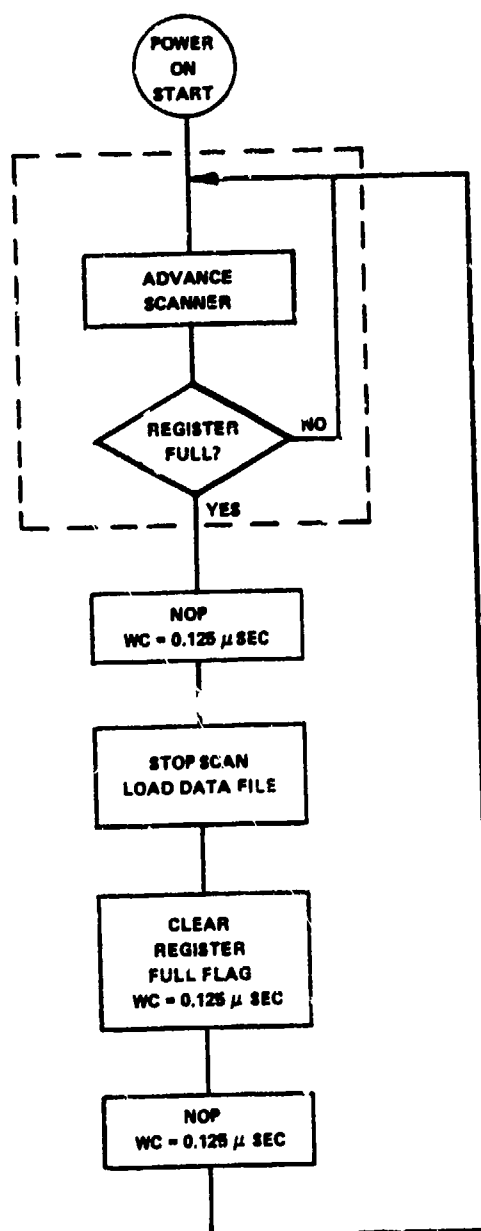
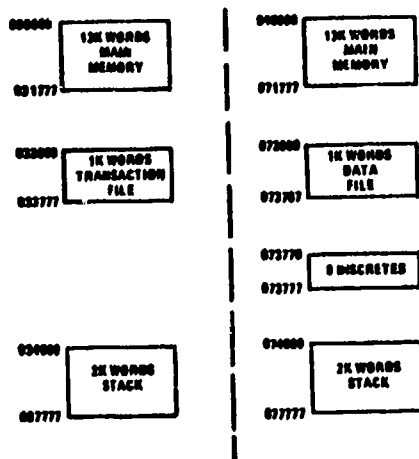
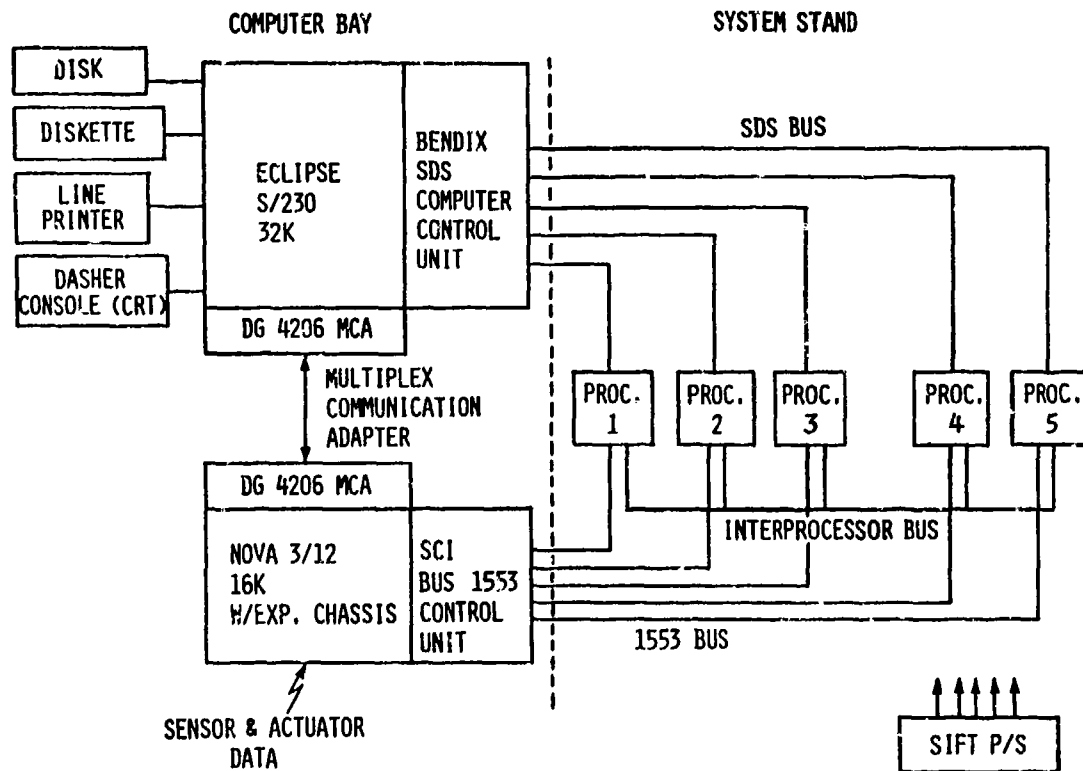
RECEIVER
SEQUENCER

FIGURE 5

MEMORY MAP
FIGURE 6

	READ	WRITE
073770	READ PROCESSOR ID	SET SNF
073771	READ 1063A STATUS REG	LOAD TRANSACTION POINTER
073772	UNUSED	
073773	READ REAL TIME CLOCK	WRITE REAL TIME CLOCK
073774	READ 1063A CMD. REG.	WRITE 1063A TEST
073775	READ 1063A CMD. REG.	WRITE 1063A CMD. REG.
073776	READ 1063A T REG.	WRITE 1063A T REG.
073777	READ 1063A T REG.	WRITE 1063A ADDRESS REG.

DISCRETES
FIGURE 7



TEST SYSTEM CONFIGURATION

FIGURE 8



FIGURE 9 SIFT LABORATORY SET-UP

STATE-OF-THE-ART COMPUTER MONITORING EQUIPMENT

Harvey G. Nelson
Naval Weapons Center
Facility Engineering Branch (Code 3115)
China Lake, CA 93555, U.S.A.

SUMMARY

In any tactical airborne computing system, it is crucial for developers and maintenance personnel to know in considerable detail what is happening inside the computer on a real-time basis. This is especially true for a distributed system. This paper describes a hardware monitor, called SOVAC (Software Validation And Control), that provides a high-capacity, real-time, user-selective "window" that gives visibility into the internal workings of the tactical computer.

1. INTRODUCTION

SOVAC is a computer monitor and controller that can be thought of in terms of its basic components and the environment it is used in. Figure 1.1 illustrates this concept.

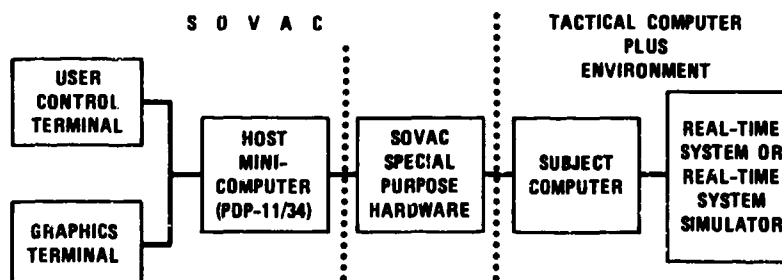


FIGURE 1.1. The SOVAC system and its environment.

There are three major components of the system: (1) The tactical computer and its environment. The minimum requirement is to have an operating tactical computer. The computer may be installed in an operational aircraft, or installed in an aircraft simulator or on a test bench. (2) The special-purpose hardware, which will be described in section 4 is attached to the tactical computer AGE (Automatic Ground Equipment) port. This allows it to monitor and, if desired, control the tactical computer's internal operation. (3) Connected to the special-purpose hardware is a general-purpose minicomputer with a cathode-ray-tube-type terminal. Our systems are using a Digital Equipment Corporation PDP-11/34 with a VT100 terminal. If hardcopy and/or graphics is desired, a suitable terminal can be added. This host mini-computer with its special-purpose SOVAC software provides the user interface to the SOVAC system and through its use interface to the tactical computer.

The basic functions of SOVAC are: (1) Automated computer control. This includes the ability to start, stop, and reset the computer. The user may also load (or read) memory or a part of memory. (2) Computer imaging. A copy of the contents of all internal registers is maintained at all times. (3) Data compare. SOVAC provides the user with the ability to take action based on the value of a specific piece of data. (4) Counting. SOVAC can count the number of times a specific event takes place. (5) Timing. The time between any two events can be accurately measured. (6) Breakpoint/event detection. SOVAC can detect a user-defined condition such as the access of a specific address location, counter value, or data value (or combination thereof) and then take a user-specified action. (7) Data selection and logging. The user may selectively cause a large number of data words and/or registers to be logged each computer cycle. (8) Tracing. Four types of traces with three triggering modes are available under user control.

SOVAC is a powerful tool for anyone who has a need to know what is happening inside a tactical computer. It is designed to be a common tool, with an absolute minimum of user-related differences between its use on various tactical computers. A key feature of SOVAC is that it is entirely passive in its effect on the target computer unless the operator specifically dictates otherwise. Thus, SOVAC provides a flexible, real-time, user-interactive tool that can greatly increase the productivity of those who work with tactical computers.

In this paper, I will review the basic steps of software engineering as applied to the life cycle of tactical software. After establishing a common reference point, I will discuss the uses of a computer monitor and display device like SOVAC in each of the phases of the software cycle. Following that, I will explain in greater detail the SOVAC functions and then briefly describe its architecture.

2. SOFTWARE ENGINEERING REVIEW

2.1. Software System Life Cycle Overview

In this section, I will discuss very briefly the life cycle process associated with tactical airborne computing systems. Most of my comments can be applied to the whole system, that is, to the hardware and software. However, I will be directing my comments primarily toward the software issues. Figure 2.1 illustrates the software life cycle process (Jensen and Tonies, 1979). By life cycle process, I am referring to the life of the system from its conception through development and its operational use, including maintenance and updates.

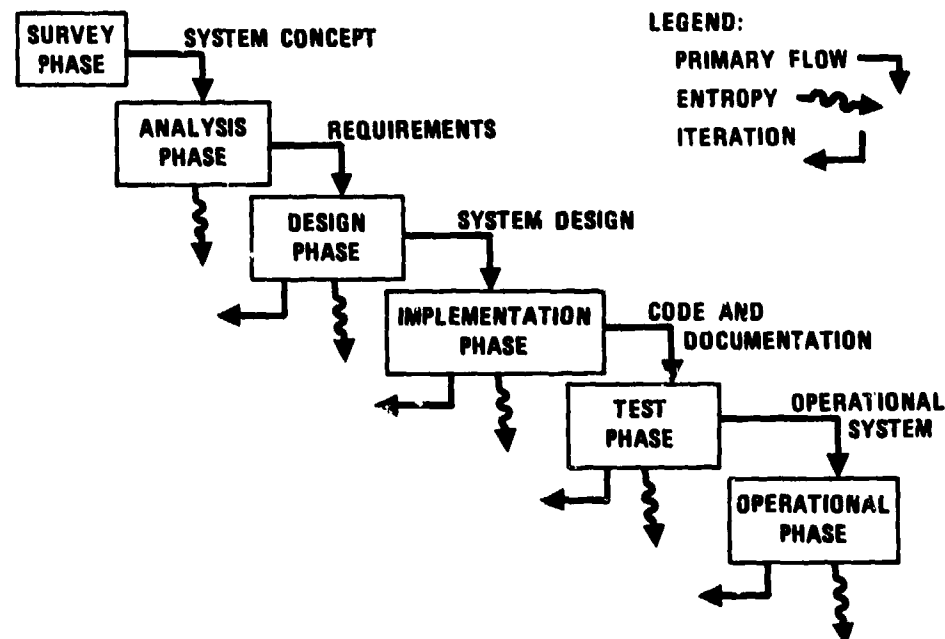


FIGURE 2.1. Software system life cycle.

I will briefly discuss each phase of the software life cycle in the following paragraphs:

2.1.1. Survey. Before one gets into a serious analysis, it is important to determine if there is justification for proceeding with a project and to document the related parameters. The survey phase is much like the analysis phase, in that it tries to define what will be built and to produce a tentative budget and schedule information. However, in the survey phase the tasks accomplished are done with the minimum of rigor needed. The output of this process is the feasibility document. In simple projects, the survey may be very brief.

2.1.2. Analysis. This process is quite involved and extremely important. It is in the analysis phase that the in-depth analysis of what is desired is documented. The analyst is responsible for understanding the needs and desires of the user and for applying his analysis tools to derive the specification. The specification should completely, but in an easy-to-understand form, define what the desired system should be able to do.

2.1.3. Design. The design process determines how the desired system will be implemented. It concerns itself with how the various functions are to be allocated among various modules and what the resultant modular interfaces will be. The modules are then designed by incorporating detailed specification information into a set of module descriptions. Finally, in the packaging process, the environment-independent design is modified to take into account the realities of the machines, operating systems, coding languages, data base processors, and so forth. In addition to the packaged design, the test plan is also generated at this stage.

2.1.4. Implementation. After the design phase, we arrive at the coding phase. I am also including in this phase the testing associated with each module before it is integrated with the other modules of the system and the integration of the individual modules into a complete package. This phase may also include any hardware/software integration required. I am including integration in the implementation phase because it is often done concurrently with the coding. The successful conclusion of this phase is an integrated program that is ready for final testing.

2.1.5. Testing. In the testing phase, the total system is tested to ensure that the requirements as derived in the analysis phase have been met and that the system is ready for operational use. This is a very crucial step for tactical systems for obvious reasons. It is also an extremely difficult task in that it is impossible to completely test even a relatively simple computer program. The result of successfully completing this phase is that the system is ready for operational use.

2.1.6. Iteration. It would be nice if we could completely finish each phase correctly before proceeding to the next. However, it seldom, if ever, works out that way. For example, in determining how to implement a specific requirement, may find that the requirement is unreasonable or perhaps impossible. Thus, formally or informally, the requirement is modified. This iteration is shown in Figure 2.1 as the arrow coming out of the bottom of the box and turning to the left. It is extremely important to provide for and control the iteration process rather than to ignore it.

2.1.7. Entropy. Entropy is the energy that is dissipated during each process and thus does not show up in the final output of the phase. Entropy is symbolized in Figure 2.1 by the squiggly arrow. Entropy is a waste of resources. In our environment, entropy refers to a waste of programmer manpower, slipped schedules, cost overruns, and perhaps failure of the project. It is this issue relative to the processes that brings us to the subject of tools for software life cycle use.

2.1.8. General Comments. My purpose in presenting the above breakdown of the process is to provide us with a common view of the process. This will facilitate the discussion that follows.

2.2. Software Tools

Bell Laboratories has established the concept of a Programmers Workbench. As Bell Laboratories uses the term, it is a collection of programs integrated with an enlightened operating system (Kernighan and Plauger, 1976). The integrated set of programs that can be used to help the programmer perform his work are called software tools. SOVAC is a tool that is composed of more than just programs.

2.3. SOVAC

2.3.1. As noted in section 1, SOVAC is a tool that is composed of a general-purpose minicomputer and special-purpose hardware and software. It is a tool that provides the user with very powerful computer control and monitoring capabilities. As a tool, SOVAC is most important in the implementation, test, and operational phase, as discussed in section 2.1 above. As will be shown later, it also is applicable to the survey, analysis, and design phases.

2.3.2. In this section I will discuss the uses of SOVAC in the context of the above discussion of the software life cycle. For each phase, I will suggest ways that we can decrease the entropy of the process by using a tool such as SOVAC.

2.3.2.1. Analysis Phase. In this phase it is especially important to know what is being asked for. As mentioned above, SOVAC has the ability to measure or quantify most information that one desires to know about the internal workings of a program in a tactical computer. Thus, if there is a prototype or earlier version of the system, SOVAC can be used to provide benchmark information such that there can be higher confidence in the resultant requirements document.

2.3.2.2. Design Phase. The ability to quantify the operation of a previous version, or a prototype, of the software can provide valuable information for the designer. In addition, the designer could use SOVAC to modify or insert a special algorithm into the target computer for evaluation.

2.3.2.3. Implementation Phase. SOVAC can be used in many ways during the implementation phase. It is in this phase that concern is primarily with the validity of a particular module or perhaps a small group of modules. SOVAC could be used to load these into the tactical computer with or without supporting modules. One could then single-step through the module, collect data, and ensure that on a stand-alone basis the module works as expected.

2.3.2.4. Test Phase. In this phase, the ability to monitor the operation of the computer without affecting its operation is also crucial. It has been shown that the ability to exhaustively test all combinations of paths through even relatively small programs is impossible. SOVAC may be used to allow much greater internal data about the operation of the program to be collected and thus gain a confidence about the program as a whole that may be very difficult to obtain otherwise.

2.3.2.5. Operational Phase. In this phase, the ability of SOVAC to monitor and collect data about the internal operation of the tactical computer without affecting its operation is crucial. SOVAC can be used to collect the data necessary to assess the validity of the operational program. As the program moves toward the need for being updated in either a major or minor manner, SOVAC can be a very useful tool to collect the data needed for the decision-making process. This ability is discussed further in the comments on the analysis and design phases.

2.4. SOVAC as a Hardware and System Tool

SOVAC is also an indispensable tool for avionics systems personnel and the hardware engineers and technicians. With the SOVAC, they can monitor and record the input and output activity. For example, it could be used to determine that a certain bit is always 0 or 1. For output channel testing, the SOVAC could be used to force a particular bit pattern on a specific output channel. Systems personnel could use it in much the same way to find out what type of data is being passed around the system. This could be especially useful with distributed systems.

3. SOVAC FUNCTIONS

In this section I will explain in moderate detail each of the basic SOVAC functions. The basic functions of SOVAC are summarized in Figure 3.1.

3.1. Automated Computer Control

Automated control includes the ability to start, stop, and reset the computer. Also, it is possible to load (or read) memory or a part of memory. Examples:

Load from a file all (or part) of memory.

Verify all (or part) of the computer memory against any specified file.

Copy from computer memory to disk, display, or hardcopy all or part of memory in hex, ASCII, binary, or real format.

Reset and start the computer at the program entry point.

Halt the computer based on any user specified condition or breakpoint.

Interactively observe and/or change any memory or register value.

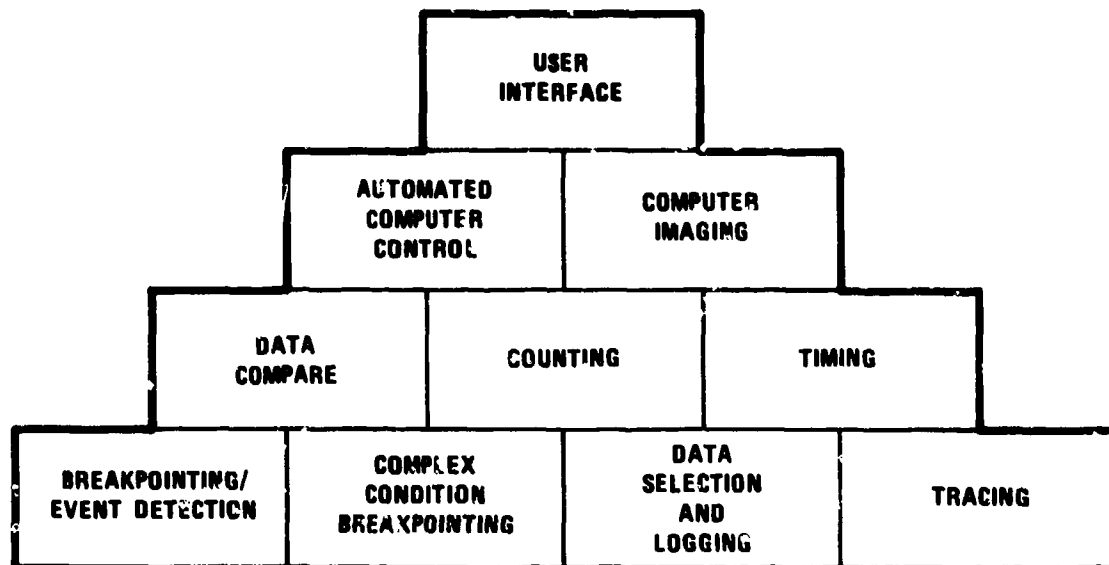


FIGURE 3.1 The ten basic functions of SOVAC.

3.2. Computer Imaging

A copy of the contents of all internal registers is maintained at all times. The contents of these image registers may be used for real-time logging and display purposes.

3.3. Data Compare

SOVAC provides the user with the ability to take action based on the value of a specific piece of data. The compare may be done against any memory location or specified register.

3.4. Counting

At the occurrence of any specified event, one of several counters may be reset, incremented, decremented, or read. This allows SOVAC to collect data on the number of times a specific event takes place.

3.5. Timing

At the occurrence of any specified event, one of several timers may be reset, started, stopped, or read. Thus, the time between any two events can be accurately measured.

3.6. Breakpoint/Event Detection

SOVAC can detect a user-defined condition such as the access of a specific address location, counter value, or data value and then take a user-specified action.

3.7. Complex Condition Breakpointing

The user may select a complex combination of events described in the paragraphs above and use these to initiate the breakpoint.

3.8. Data Selection and Logging

The user may selectively cause a large number of data words and/or registers to be logged each computer cycle.

3.9. Tracing

Four types of traces with three triggering modes are available under user control. For each type of trace, the user-specified data is stored in a first-in-first-out history (FIFO) stack. This is done on a real-time basis without altering the operation of the tactical computer.

The types of traces available include:

Full instruction trace where each instruction cycle initiates the storage into the FIFO stack.

Partial instruction trace where each instruction within a user-specified range is recorded. This allows concentration of the data gathering resources to a specific area of interest.

Event trace allows data to be stored at each occurrence of a specific event.

Branch trace allows the storage to take place when the instruction counter changes by greater than a user-specified value.

The three modes of the trace function are:

Normal. In this mode the trace action takes place when the breakpoint occurs.

Delay. In this mode the trace action is delayed from the specified event by a user-selectable time, count, or number of events.

Prerecord. In this mode the trace action takes place at each specified event and is stopped by a user-specified event. This allows the operations up to a specific event to be stored. Adding a user-specified delay to this mode allows the operations before and after the specified event to be recorded.

4. SOVAC ARCHITECTURE

4.1. Overview

The basic functional components of SOVAC are shown in Figure 4.1 below.

The following sections will explain the role of each part of the SOVAC hardware in further detail.

4.2. Tactical Computer Interface

The tactical computer interface is composed of the computer control section and the computer image section. The computer control section provides real-time control of the tactical computer and provides the capability to capture information available on the tactical computer's bus and control lines. This is the most difficult of the sections to design. In general we have found it very hard to obtain the level of documentation necessary to make the design straightforward. The techniques used have combined detailed analysis of the computer documentation available and the use of logic analyzers to gather empirical data about the activity on the AGE port. It also happens that not all desired signals are available at the AGE port. This makes the SOVAC design much more difficult in that the internal activity of the computer must be inferred from the activity of the signals that are available.

The computer image section contains a copy of each of the tactical computer's registers. For those situations in which a given register may have more than one function during the execution of a specific instruction, a copy of the register for each of its functions is provided. Thus, at any time, the contents of all the internal registers are available for whatever use is desired. These uses will be discussed in the following sections.

4.3. SOVAC Controller

The high-speed, microprogrammed SOVAC controller coordinates the operation of the various subsystems. It monitors the contents of the tactical computer image registers and has the capability to recognize various types of events or complex combinations of events and set a breakpoint or store data into a hardware FIFO stack that is 18 words wide. The data selection and logging capability is very flexible. The breakpoints can be used for a wide variety of purposes including the control of counters and timers.

The functions of the controller are set up under the control of the user. The controller must do all of its evolutions within the instruction cycle time of the tactical computer. Thus, the activity in the controller is several orders of magnitude faster than could be controlled from the host minicomputer. Therefore, the controller functions are set up by the host minicomputer under user control.

4.4. Minicomputer Interface

This interface is placed on the minicomputer system bus and provides the data path between the minicomputer and the SOVAC controller.

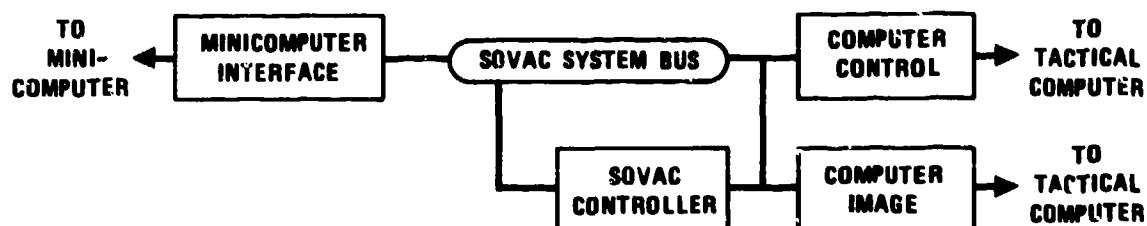


FIGURE 4.1. SOVAC hardware system.

4.5. Commonality/Portability

This is an appropriate time to note that the SOVAC concept is not applicable to just one specific tactical computer. In fact, considerable effort has been expended to minimize the uniqueness of each SOVAC. This is especially true of the user interface. If a user knows how to use a SOVAC for one tactical computer, he should know how to use it for any others without further training. For maintenance purposes it is also desirable to minimize the differences between the hardware designs of the various SOVAC models. Referring to Figure 4.1, the minicomputer interfaces (and the minicomputers and peripherals) are common among all SOVACs. Also, the bus systems are common between the SOVACs. The computer control and computer imaging hardware is unique to each tactical computer type. The event detection, selection and logging, and system controller are similar in concept but unique in implementation due to the differences in the number and type of registers in the various computers.

5. CONCLUSION

SOVAC is a powerful tool for anyone who has a need to know what is happening inside a tactical computer. It is designed to be a common tool with an absolute minimum of user-related differences between its use on various tactical computers. A key feature of SOVAC is that it is entirely passive in its effect on the target computer unless the operator specifically dictates otherwise. Thus, SOVAC provides a flexible, real-time, user-interactive tool that can greatly increase the productivity of those who work with tactical computers.

REFERENCES/BIBLIOGRAPHY

- DeMARCO, Tom, 1978, "Structured Analysis and System Specification", Yourdon Press.
- DeMARCO, Tom, 1979, "Concise Notes on Software Engineering", Yourdon Press.
- FLETCHER, W. L., 1980, "An Engineering Approach to Digital Design", Prentice-Hall.
- JENSEN, R. W., and TONIES, C. C., 1979, "Software Engineering", Prentice-Hall.
- KERNIGHAN, B. W., and PLAUGER, P. J., 1976, "Software Tools", Addison-Wesley.
- FRYER, R. E., 1980, "The User Interface for a Real-Time Software Debugging System", Proc. of the Fourteenth Asilomar Conference on Circuits, Systems, and Computers.
- LEMON, L. M., 1979, "Hardware System for Developing and Validating Software", Proc. of the Thirteenth Asilomar Conference on Circuits, Systems, and Computers.
- YOURDON, E., 1979, "Classics in Software Engineering", Yourdon Press.
- YOURDON, E., and CONSTANTINE, L. L., 1979, "Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design", Prentice-Hall.

INTEGRATED CONTROL OF MECHANICAL SYSTEMS FOR
FUTURE COMBAT AIRCRAFT

G. W. WILCOCK

Royal Aircraft Establishment, Farnborough, U.K.

P. A. LANCASTER & C. MOXEY

British Aerospace, Warton Division, U.K.

SUMMARY

This paper describes a system for achieving digital control and monitoring of Utility Systems for future combat aircraft. The aim is to:

- i) Reduce penalties such as mass and engine power take-off associated with conventional systems.
- ii) Reduce pilot workload.
- iii) Improve maintainability.
- iv) Increase survivability.
- v) Reduce the cost of ownership.

The paper explores various approaches to system design, leading to a system utilising distributed processors and data terminals linked via interfaces to the Utility Systems' components.

The work to date has shown that a significant number of the objectives can be achieved; for example, a weight saving of approximately 100 Kg (i.e. 50%), and a pilot workload reduction of the order of 4:1, may be achieved in a twin engine combat aircraft.

1. INTRODUCTION

In today's generation of combat aircraft, mechanical systems or "Utility Systems" - such as those associated with Powerplant Control, Secondary Power, Environmental Control, Hydraulics and Fuel Gauging/Management - have been designed as individual systems and consequently have their own dedicated control units. The result is:

1. A large number of dedicated, single function Line Replaceable Units (LRUs).
2. Boxes containing relay and diode logic.
3. Large numbers of discrete wires.
4. Many dedicated cockpit instruments.
5. Dedicated switches and warning lamps.

The many interconnections result in large cable looms; these impose a severe installation penalty on the aircraft. The cable looms may also be subject to damage and Electro-Magnetic Interference.

For future aircraft this method of control is likely to be unsatisfactory due to the limited space available and is inefficient in terms of equipment utilisation. In addition, future high technology combat aircraft will incorporate a highly integrated avionic system and will require increased automation from all systems, including Utility Systems, in order to significantly reduce the pilot workload (especially in a single cockpit configuration).

The above demands, together with the increasing use of serial digital data transmission systems means that alternative design methods must be applied to Utility Systems.

A considerable amount of research work has been progressing at British Aerospace, Warton (under both MCD and Private Venture funding) and also at the Royal Aircraft Establishment, Farnborough, into alternative methods of controlling the Utility Systems. The most favoured approach for realising this control is to consider a Central Management System which controls ALL of the Utility Systems, as listed in Para. 3.

The result of this approach is the Integrated Control of Mechanical Systems (INCOMS) which is based on a number of data acquisition and control units (INCOMS Processors) which are geographically dispersed throughout the airframe. These INCOMS Processors will operate independently as individual computing centres, and will be interconnected via a MIL-STD 1553B data bus (or its derivative). Some of these Processors will act as remote terminals, collecting raw data for onward transmission (via the data bus) to their designated processing centre(s).

It must be borne in mind that, whilst most Utility Components are inherently simplex in nature, (in terms of their input and output functions), the systems in which they are incorporated are mainly safety critical systems. Separate redundant components and control circuits are included, where appropriate, to achieve the necessary reliability.

2. BACKGROUND SUPPORT ARGUMENT

Various reasons can be cited to support the view that digital control should be used (Seabridge, A. G., 1979; Smith, T. B., et al, 1978). The general arguments for supporting the implementation of digital control for utility systems (INCOMS) are twofold, namely:

- a) Safety.
- b) Efficiency.
- a) SAFETY improvements can be realised by:
 - i) System integrity improvements through increased reliability and the more efficient use of redundancy techniques.
 - ii) Reduction of the pilot workload as a result of increased system automation.
 - iii) Improved communication capability (between the sub-systems).
- b) EFFICIENCY improvements are expected due to:
 - i) Reduced wiring and reduced weight.
 - ii) Easing of the maintenance task by providing a self-test and fault diagnosis capability. This would provide status indications to flight and ground crews to establish pilot confidence in correct system operation (even under fault conditions). It would also reduce the maintenance time.
 - iii) Reduction in the initial and life cycle costs.
 - iv) Improved System Performance leading to load scheduling and, hence, more efficient use of available power.

A further argument to support the general philosophy is that the introduction of digital control allows the system to be designed so that, even in service, but particularly during development, advantage can be taken of technological advance. In addition the system can be adapted at reasonable cost to meet changing demands. (Durkin, H., 1977).

On today's aircraft the majority of control changes tend to be hardware orientated, whereas in INCOMS it is anticipated that most changes can be implemented in software, thereby reducing the impact on the airframe.

The adoption of this type of system will give a reduction in the number of dedicated equipments, thereby reducing overall equipment mass and volume; and should promote the design and use of common items of hardware.

INCOMS will have a significant impact on the crew-system interface and thus upon the requirements for cockpit displays and controls. A fully automatic management system with the ability to operate without significant degradation under fault conditions will reduce the necessity to continuously display status information. A digital computing system with access to the cockpit via MIL-STD 1553B will enable the pilot to communicate with systems via non-dedicated or multifunction switches.

3. EVOLUTION OF THE INTEGRATED CONTROL OF MECHANICAL SYSTEMS (INCOMS) PHILOSOPHY

The systems that are being considered in the global term "Utility Systems" are shown below:

Engine and Associated Systems

- | | |
|----------------------------------|--------------------------------|
| * Engine Intake De-Icing | * Engine Starting and Ignition |
| * Engine Speed Signals | * Thrust Reverse Control |
| * Fire Detection and Suppression | * System Health Monitoring |
| * System Warnings | |

Hydraulic and Associated Systems

- | | |
|------------------------------|--------------------------------|
| * Hydraulic Utilities | * Hydraulic Control |
| * Hydraulic Depressurisation | * Brakes and Anti-Skid Control |
| * Undercarriage Control | * Nosewheel Steering |
| * Flight Refuelling Probe | * Canopy Control |
| * Systems Health Monitoring | * System Warnings |

Fuel System

- * Fuel Management
- * Re/Defuel Transfer
- * Fuel-Flow Metering
- * System Health Monitoring
- * Fuel Boost Pump Control
- * Fuel Gauging
- * Hit Detection and Suppression
- * System Warnings

Oxygen Supply System

- * Nuclear/Chemical/Biological Protection

Environmental Control Systems

- * Cabin Temperature Control
- * Rain Dispersal
- * Equipment Bay Cooling
- * System Health Monitoring
- * Temperature & Pressure Safety Control
- * Canopy Standby De-Mist
- * Coolant System Control
- * System Warnings

Secondary Power System

- * Gearbox Control
- * Emergency Power Unit Control
- * Auxiliary Power Unit Control

Miscellaneous Systems

- * Cockpit Lighting
- * Landing and Taxi Lights
- * Windscreen Heating Control
- * Seat Adjustment
- * Electro-Luminescent Panel Control
- * Anti-Collision Lights
- * Probe Heating
- * Arrestor Hook

The systems range from the very complex, e.g. Fuel Management, to the very simple, e.g. Arrestor Hook. The one thing that all of these systems have in common is that they must conform to the aims and constraints of INCOMS.

A brief description of the "total systems" approach that has been adopted at BAe, Warton and at RAE, together with a description of an ideal system follows, to give some background to the technical aspects of the work carried out to date.

Figure 1 shows a block diagram of a typical integrated avionic system envisaged for future aircraft, but with the Utility systems shown as they exist on contemporary aircraft - i.e. JAGUAR/TORNADO. These Utility systems have individual sets of components and control elements. Only five control elements are shown, whereas on today's aircraft one set of control hardware would be expected for each of the systems. To connect the individual systems to the cockpit would require a considerable amount of discrete wiring and would be contrary to the general policy of using digital data transmission.

To avoid this situation, all the control elements could be combined into a single block called Utility Systems Management (see Figure 2), and that block connected as a terminal onto the main Avionics Bus. If recognition is taken of the distribution of Utility Systems components throughout the airframe, it will be seen that this method is unacceptable for at least three reasons:

- a) The large amount of discrete wiring involved.
- b) The concentration of wiring at the central block.
- c) The susceptibility of the central block to damage or failure.

These problems can be overcome by the Integrated Control of Mechanical Systems (INCOMS) whose system consists of a number of data acquisition and control devices situated at strategic locations throughout the airframe (see Figure 3).

This will enable components local to the devices to be connected to the most suitable (e.g. nearest) device, thereby restricting discrete wiring to local areas. A data acquisition and computing sub-system can now be considered which consists of a number of INCOMS processors (the current work at BAe, Warton indicates that 6 may be an optimum number) which are interconnected via a MIL-STD 1553B data bus. This bus is in turn connected to the main Avionics Bus via Bus Interface Units (BIFU) which can also act as Bus Controllers. This is illustrated in Figure 4.

The interfaces with the data bus, CPU and memory, and the interfaces with the mechanical system's components are shown. These interfaces will allow receipt of information from discrete, analogue, digital and optical devices/sensors, and power control interfaces will allow power to be switched to devices such as valves, pumps, etc. In an ideal system each INCOMS processor would be hardware identical, with its individual program store taking account of the various peripheral components in each INCOMS processor's aircraft location.

This system offers a minimum hardware, minimum wiring solution that can be envisaged for a future combat aircraft.

Work already completed by British Aerospace, Warton under UK MOD funding, tested the above philosophy applied to a possible future advanced fuel management system. The aim of this work was to define a suitable overall system architecture that would give a management system in which greater emphasis is placed on automation, fault detection/tolerance and survival. In pursuance of this a number of possible configurations were considered.

In each of the configurations studied, control was assumed to be based on some form of digital computing system, installed in a twin engined aircraft. The configurations ranged from a system employing discrete, dedicated wiring between components and computer (Kaye, A., 1979; Moxey, C., 1980) to those employing a distributed processing system with a serial data bus connection (Seabridge, A. G., 1980).

Each configuration was tested against a representative aircraft layout, and it was observed that some of the earlier configurations presented major wiring problems, in that they were complicated to install and also left major sections of the system susceptible to battle damage. To overcome these basic problems the later configurations subjected the system to further detailed examination in order to determine the optimum interconnection of components and computing system elements that satisfy the conditions of fault tolerance and serviceability, whilst also being possible to install.

In order to meet the overall integrity requirements of the systems included in INCOMS it was considered necessary to develop a method to define the relative importance of these systems and prepare from this a Criticality Analysis and Ranking List (CARL). Information thus obtained would be used to identify if a need exists to duplicate data, to cross-strap from sensors to computing elements by hard wiring, or if a pre-set datum needs to be introduced in the event of failure of primary data sources.

To this end four methods were investigated, each taking account of the effects of systems failure on flight or mission success, in both peace time and war time operation. The first three methods considered were all discounted (Lancaster, P.A., 1980). By adoption of the fourth method the CARL chart as shown in TABLE I was compiled. Briefly, this method was developed through discussions with engineers whose experience spanned a number of aircraft projects, where the INCOMS systems were broken down into five discrete levels in order of importance, see TABLE II. Also taken into account was the frequency of each system operation during the different phases of a mission, see TABLE III. A full description of this analysis may be found in Lancaster, P. A., 1980.

SYSTEM	RANKING	SYSTEM	RANKING
FUEL SYSTEM	130	CABIN TEMPERATURE CONTROL	54
HYDRAULICS-CONTROL	125	NOSE WHEEL STEERING	52
GEAR BOX CONTROL	120	DEPRESSURISATION	52
IGNITION MANAGEMENT	105	BRAKE/ANTI-SKID	45
ENGINE CONTROL SERV	100	WINDSCREEN HEATING	28
HYDRAULICS UTILITIES	100	COCKPIT LIGHTING	28
APU /EPU	100	THRUST REVERSE	28
SYSTEMS WARNINGS	95	ARRESTOR HOOK	26
OXYGEN	92	CANOPY DE-MIST	20
U/C CONTROLS AND IND.	90	HIT DETECTION	16
EQUIP. BAY COOLING	84	HEALTH MON/MAINT	15
N.B.C.	80	RAIN DISPERSAL	15
AIR SYSTEM CONTROL	72	CANOPY CONTROL	14
CABIN ALTITUDE	69	ANTI-COLLISION LIGHTS	13
REFUEL PROBE	68	NAV/OBST LIGHTS	12
PROBE HEATING	64	E.L. PANELS	10
FIRE DETECTION	60	LAND/TAXI LIGHTS	10
INTAKE DE-ICING	57	SEAT ADJUSTMENT	3
ENGINE START	55		

TABLE I -- CRITICALITY ANALYSIS RANKING LIST (CARL)

LEVEL	SYSTEMS	
1	FUEL MANAGEMENT ENGINE START IGNITION MANAGEMENT ENGINE CONTROL SERVICES GEAR BOX CONTROL	APU/EPU HYDRAULICS-CONTROLS UNDERCARRIAGE NBC SYSTEMS WARNINGS
2	FIRE DETECTION HYDRAULIC UTILITIES DEPRESSURISATION NOSE WHEEL STEERING PROBE HEATING	AIR SYSTEM REFUEL PROBE HIT DETECTION EQUIP. BAY COOLING OXYGEN
3	INTAKE DE-ICING BRAKES AND ANTI-SKID CABIN ALTITUDE	HEALTH MON./MAINT. RECORDING CABIN TEMP CONTROL
4	WINDSCREEN HEATING COCKPIT LIGHTING CANOPY DE-MIST	ARRESTOR HOOK E.L. PANEL THRUST REVERSE
5	CANOPY CONTROL RAIN DISPERSAL LAND/TAXI LIGHTS	ANTI-COLLISION LIGHTS NAV./OBST. LIGHTS SEAT ADJUSTMENT

TABLE II - CRITICALITY LEVELS

SYSTEM	START	TAXI	TAKE -OFF	CRUISE	COMBAT	APPR & LAND.
ENG. IGNITION	-----	*****	*****	*****	*****	*****
ENG. START	-----	-----	-----	-----	-----	-----
ENG. CONT. SERV (R.P.M.)	-----	-----	-----	-----	-----	-----
ENG. CONTROL	-----	-----	-----	-----	-----	-----
ENG. INTAKE DE-ICING	-----	-----	-----	-----	-----	-----
THRUST REVERSE	-----	-----	-----	-----	-----	-----
FIRE DETECTION AND SUPP.	XXXXXX	XXXXXX	XXXXXX	XXXXXX	XXXXXX	XXXXXX
ENG. GEAR & X DRIVE	-----	-----	-----	-----	-----	-----
APU/EPU	-----	-----	-----	-----	-----	-----
HYD. UTL.	-----	-----	-----	-----	-----	-----
P.Y.D. CONT.	-----	-----	-----	-----	-----	-----
DEPRESSURISATION	-----	*	-----	*****	*****	-----
BRAKES AND ANTI-SKID	-----	-----	-----	-----	-----	-----
CANOPY CONTROL	-----	-----	-----	-----	-----	-----
U/C CONT. AND IND.	-----	-----	-----	-----	-----	-----
NOSE WHEEL STEERING	-----	-----	-----	-----	-----	-----
FLIGHT REFUELLING PROBE	-----	-----	-----	-----	-----	-----
FUEL BOOST PUMPS	-----	-----	-----	-----	-----	-----
LP COCKS	-----	-----	-----	-----	-----	-----
RE/DEFUEL AND TRANS.	-----	-----	-----	-----	-----	-----
FUEL DUMP	-----	-----	-----	-----	-----	-----
FUEL GAUGING	-----	-----	-----	-----	-----	-----
FUEL FLOWMETERING	-----	-----	-----	-----	-----	-----
HIT DET. AND SUPP.	XXXXXX	XXXXXX	XXXXXX	XXXXXX	XXXXXX	XXXXXX
CABIN TEMP. CONTROL	-----	-----	-----	-----	-----	-----
AIR SYS. CONT.	-----	-----	-----	-----	-----	-----
RAIN DISPERSAL	-----	-----	-----	-----	-----	-----
CANOPY DE-MIST	-----	-----	-----	-----	-----	-----
EQUIP. BAY COOLING	-----	-----	-----	-----	-----	-----
N.B.C.	-----	-----	-----	-----	-----	-----
COCKPIT LIGHTING	-----	-----	-----	-----	-----	-----
E.L. PANEL	-----	-----	-----	-----	-----	-----
LAND/TAXI LIGHTS	-----	-----	-----	-----	-----	-----
ANTI-COLL LIGHTS.	-----	-----	-----	-----	-----	-----
NAV/OBST LIGHTS	-----	-----	-----	-----	-----	-----
W/S HEATING	-----	-----	-----	-----	-----	-----
PROBE HEATING	-----	-----	-----	-----	-----	-----
SEAT ADJUST	-----	-----	-----	-----	-----	-----
SYSTEMS WARNINGS	-----	-----	-----	-----	-----	-----
HEALTH MON/MAINT REC.	-----	-----	-----	-----	-----	-----
CABIN ALTITUDE	-----	-----	-----	-----	-----	-----
OXYGEN	-----	-----	-----	-----	-----	-----

KEY: ----- Continuous Operation; ***** Possible Systems Requirement
XXXXXX Continuous Monitoring

TABLE III - SYSTEM UTILISATION DURING A MISSION

The analysis gives an indicator of the integrity targets that must be met for the individual systems, and for the total INCOMS system. In conjunction with the geographical position of the sensors and actuators it also helps in the allocation of tasks to individual processors. As a result the number of interfaces at each Processor can be defined.

4. CONTROL SYSTEM DESIGN REQUIREMENTS

To meet the requirements of the Utility Systems, particularly in respect of reliability and integrity, without imposing intolerable burdens of mass, complexity and cost requires careful evaluation of alternative approaches to system design. It is essential to think in system level terms at the outset so that the design philosophy encompasses all components and requirements of the system. Some of the special features and constraints of the application and their influence on the major components of the control system are discussed in the following sub-sections.

4.1 Control Requirements and Characteristics

It is convenient to categorize the control tasks in Utility Systems on a hierarchical basis, as shown in TABLE IV. At the lowest level most of the data is in single bit quantities representing physical parameters such as limit-switch position or valve open-close command. Additionally, there is a lesser number of data values related to analogue quantities such as temperature and pressure. Processing at this level consists largely of packing and unpacking words for computation and transmission, evaluation of Boolean expressions for load control, and the generation of status information for individual Utility components and their interfaces.

SYSTEM EXECUTIVE (System functions, status, bus control)			
FUEL EXECUTIVE	HYDRAULICS EXECUTIVE	SUB-SYSTEM EXECUTIVES (Mode Selection, Status Monitoring)	SEAT ADJUSTMENT EXECUTIVE
TRANSFER, ENGINE RE-LIGHT, QUANTITY CALD. ETC.	ENGINE START, PRESSURE SCHEDULING ETC.	SUB-SYSTEM FUNCTIONS	RAISE/ LOWER INTERLOCK
VLV/PUMP CONTROL, LEVEL, FLOWRATE SENSING ETC.	VALVE CONTROL, FLUID LEVEL SENSING ETC.	COMPONENT LEVEL FUNCTIONS	POWER SWITCHING TO MOTOR

TABLE IV - CLASSIFICATION OF UTILITY SYSTEM CONTROL TASKS

At the next level the basic functions or operating modes of the Utility sub-systems are accomplished. For example, in the fuel system some of the modes would be fuel quantity computation, refuel/defuel, transfer, engine start-up, engine relight and system status monitoring. Most functions at these levels would involve logical processes and sequences with a lesser number of analogue functions such as control of braking deceleration. Each of the functions may either be fixed or one of a set of related functions to cover different flight regimes, aircraft status, or failure modes.

The selection of individual modes would be made by the system executive in response to pilot commands and status data. Practical considerations indicate the desirability of splitting this executive function into two levels. The lowest of these extends as far as the sub-system boundary so that a vertical structure is now imposed on the functional diagram (TABLE IV) to separate sub-systems into distinct modules. This both eases the design process and improves visibility and integrity, and accords with techniques for structured system design. The main functions of the topmost executive level would now be to instruct individual sub-system executives as to the desired mode of operation, handle data flow in the system and perform global functions such as load shedding or engine relight which require action in a number of sub-systems. In existing aircraft most of the executive functions are performed by the pilot, in addition to many of the sequences and some of the individual component level switching and monitoring actions. Replacement

by computer control will obviously reduce pilot workload significantly, although a system which leaves the pilot without control is neither desirable nor acceptable. Means must be provided to enable the pilot to monitor system operation and to over-ride executive functions when desired.

The processing requirements of the sub-systems vary considerably from the complex (e.g. fuel) to the very simple (e.g. seat adjustment). In the case of the simplest the classification into function level may appear academic since only minimal processing is required at each level, for example to pass a discrete signal from the cockpit to the seat motor. However, there is value in retaining the classification for consistency in specification, design and implementation.

The partitioning described suggests that distributed processing could be appropriate on either a horizontal (sub-system) or a vertical (function level) basis. The design of a system which assigns groups of sub-systems to different processors, with separate units to perform the System Executive function is described later in section 5.2.

4.2 Reliability and Integrity

The critical nature of many Utility sub-systems to aircraft safety has already been underlined. The reliability of mechanical and electrical components demands the use of varying levels of redundancy to achieve sufficient safety and this translates into the control system. A simple approach would be to determine the level of redundancy required to meet the most critical needs and apply this at all levels of the system from processors to interfaces. However, this would lead to significant mass and cost penalties due partly to the large number of discrete signal sources and sinks. Consider the basic output interface function of load switching: to provide dual redundancy against open and short-circuit failures of the switch elements requires them to be quadruplicated, as shown in Figure 8. This arrangement must be used because Utility components do not in general lend themselves to a redundant configuration similar, for example, to actuators for flight control: separate actuators or sensors must be replicated to achieve redundancy. Since there may be around 200-500 discrete outputs on an aircraft the total mass penalty could be considerable with current solid-state switches having a mass of around .04 to .20 Kg depending upon rating. N-fold redundancy requires N^2 switch elements so that the application of redundancy at the discrete output level needs to be tailored to the needs of individual circuits rather than applied wholesale. The same consideration should be applied at all interface circuits since studies have shown that they dominate the mass and complexity of the local control units owing to the large number required.

The optimum solution for the Utility Systems would be to accept varying levels of redundancy throughout the system for different circuits and sub-systems and at the different function levels. Many mechanical and electrical components have failure rates of the order of 10^{-4} /hour, and for their related interfaces duplex or even simplex redundancy would be adequate. However, a basic principle of their design is that to reduce the probability of multiple failures individual sub-systems are isolated, hence higher levels of redundancy are essential for processors and data terminals which could cause multiple function loss upon failure. An acceptable level of total control failure is unlikely to be less than around 10^{-5} /mission and studies have shown (Collingbourne, L.R., 1981) that at least 3-fold redundancy will be required based on reasonable estimates of failure rates and success for BIT procedures. At the local terminal level lower redundancy levels could be acceptable since each handles only part of the system. This aspect of design involves careful analysis of Utility System operation so that signals can be grouped to avoid mutual reinforcement of disaster. This would occur, for example, if control of wheel-brakes and reverse thrust were obtained from the same terminal but not for the combination of fuel-level and reverse thrust. Finally, use can be made of reversionary modes in the event of terminal or system failures so that loads assume a preferred state, e.g. fuel pumps permanently on.

Assuming that Avionic requirements lead to the use of a duplex data bus for communication with the cockpit, a subsidiary panel and data link providing flight safety instruments and controls for the most vital Utility functions is likely to be necessary. This solution could be preferable to penalising the main Avionic bus with requirements for a higher level of redundancy.

4.3 Data System

Provision of a data system to reduce the mass and complexity of the cabling linking widely dispersed Utility components has been shown to be a prime requirement for future designs. The widespread adoption of MIL-STD1553B for national (e.g. UK Defence Standard OC-18 (Part 2)/1) and international standards for military aircraft is a powerful argument for its adoption for Utility Systems and most of the work in the UK has assumed its use. Studies have shown (Wilcock, G. W., 1978; Moir, I., 1981) that for a centralised system a bus loading of around 25-38% is likely to result. Such a relatively high level supports the argument for provision of a separate Utility System bus with an interface to the main Avionic and cockpit busses so that traffic on the latter is increased only by the smaller amount needed for cockpit control and display.

A loading of 25-38% is acceptable for the Utilities bus, although since these values were based on relatively early designs lacking many of the higher level control tasks it is a factor to be kept under careful scrutiny. A distributed processing system would tend to alleviate problems by reducing the amount of low level data on the bus. The need for a

bus-controller in MIL-STD-1553B systems tends to mitigate against a distributed approach, although it is compatible with a hierarchical system where it could be incorporated at system executive level.

A practical consideration which has a major influence on system design is the relative complexity of MIL-STD-1553B terminals, comparable in LSI chip count and failure rate with microprocessors. The implication is that the penalties of incorporating intelligence into terminals in terms of increased failure rate, mass and volume will be small. This, then, is a strong indication for distributed processing based on microprocessors within the data terminals.

4.4 Software

Although the basic control algorithms for individual Utility components are relatively simple the overall control task which must allow for fault detection, corrupt data and automatic control in a system with of the order of 500 inputs and outputs is formidable. Practical constraints of mass and complexity do not permit monitoring of mechanical and electrical Utilities to the degree necessary to uniquely identify all possible single faults. Multiple faults are less common but do occur and could be extensive following battle damage. Loss of power at one or more electrical bus-bars causes widespread corruption of data from sensors supplied from them. The task of coping with these problems falls largely to the aircrew at present. Automatic control must attempt to duplicate to some extent the partly intuitive logic processing of the aircrew to be wholly successful, although such complete control must await advances in the state of the art. Nevertheless, computer based system monitoring and automatic control designed on existing knowledge could significantly reduce pilot workload. The problem still remains that software must cope, without failure, with corrupted data and incompletely defined system status magnifying the problems of software design and validation. An approach which results in "resilient" software is desirable. An assessment of the problem (Collingbourne, L. R., 1980) has shown the value of the appropriate high-level language and a structured self-documenting approach, whilst the SAFRA methodology (Ward, A. O., 1979) has been used in trials which show that the method leads to a rigorously obtained, fully documented requirement. The use of text processing and automated aids for data consistency checking used in this method make it a valuable aid in a structured design process. A particular problem in the Utility Systems is the large number of autonomous simple functions which would lead to processing inefficiency if each were modularized. A possible solution is to group simple functions into modules, relying on the self documenting features of a language such as CORAL 66 to achieve integrity through visibility.

A distributed approach to computing would ease software problems by reducing the size of individual programs, although the provision of a validated distributed executive would be a critical task. The MASCOT system (MASCOT Suppliers Association, 1980) for multi-tasking, although mainly thought of for use on centralised systems at present, could be applied to distributed processors and would be a valuable system building tool.

5. SOME APPROACHES TO SYSTEM DESIGN

Studies are currently in progress to investigate alternative approaches to system design. The optimum choice for a particular application is affected by procurement considerations as well as technical factors and definitive answers have yet to be produced. In this section the two basic approaches of centralised and distributed processing are considered with emphasis on the latter. It was considered that in dealing with safety critical systems a pessimistic rather than an optimistic view of failure was appropriate, hence the need for relatively high levels of redundancy and integrity have been assumed. With practical experience some simplification might be achievable.

5.1 Processing Based on Parallel Redundancy

Synchronized parallel redundancy is an established technique for detecting and isolating failures with high confidence and integrity. Triple modular redundancy (TMR) is the minimum level capable of meeting the requirements of system failure rate and fault level capability. The architecture of a possible system is shown in Figure 5. The system and bus controllers and software at all function levels are triplicated and linked via a triplex bus to 6 local data terminals adjacent to Utility components. Within local units the data bus terminals and interface controllers are triplicated. Discrete inputs are cross-wired to all 3 channels with relatively little penalty in complexity, but it is sufficient to reduce to duplex redundancy in individual discrete outputs since availability requirements are less than at terminal or processor levels. There is a range of possibilities for distribution of the voting and data consolidation points within the system. That shown in Figure 5 enables failures within the interfaces and their controllers to be isolated at local level but data terminals are included as part of the bus structure. Consolidation could be included between busses and terminals increasing system availability at the expense of added complexity.

A disadvantage of TMR is the prospect of common-mode failure due to causes such as EMC, EMP, common servicing defects, generic software failures, propagation of electrical faults, or by failure of BIT arbitration after 2 failures. The relatively long time-constant of most Utilities should enable recovery from transient effects such as EMC. Electrical isolation could be achieved using fibre-optic links between channels, although at some

penalty in complexity. An alternative approach which is particularly appropriate since mechanical and electrical systems are already largely based on a two channel structure (designated left and right channels) is to use a dual duplex system, as shown in Figure 6. Isolated left and right-hand dual busses are used, with duplex redundancy within each system and bus controller, and within local terminals. Comparison of channels is used to detect failures, with BIT to determine the faulty channel. The failures are detected with high confidence, although the probability of incorrect arbitration is increased relative to TMR. However, any defects in this area are confined to one half of the system so that overall integrity and availability requirements are met. Assuming the need to communicate with a duplex Avionics bus dual links to each duplex Utility bus are needed so that a single fault does not result in loss of function. Fibre optic links are desirable to preserve electrical isolation.

Both of the systems described are complex in terms of both the number and capabilities of the functional units. Although only 3 processors are required for TMR they must each have the capability to handle total system and bus control. Mission reliability could be degraded using TMR since missions may have to be aborted following single failures since further ones could hazard flight safety. In the dual duplex arrangement vulnerability considerations are a strong indication for separation of the processors in each duplex channel, increasing the number of boxes and complexity since bus terminals would have to be duplicated for each processor rather than for pairs of processors.

5.2 Distributed System

The dual duplex system represents a stage in the progression to distributed systems since it effectively separates processing between left and right-hand channels each controlling only half of the Utility System. The next stage is to incorporate local processors within the already distributed data terminals. Although the system, shown in Figure 7, resembles the dual duplex system symbolically, its characteristics and operation are quite distinct. Its design is based on the partitioning scheme described earlier. There are practical limits on inter-system (horizontal) partitioning since it would be wasteful of resources to provide individual processors for the many sub-systems involved, i.e. 30-40, nor would it be appropriate in view of their widely differing processing requirements. A preferred approach would be to distribute groups of sub-systems to different groups of processors, the redundant functions of each sub-system being partitioned between different processors in a group. Allocation to a particular processor would involve considerations of component layout, processor utilization, and reliability requirements. Separate processors are shown to perform the function of the System Executive, with bus control (assuming a MIL-STD-1553B based data system) and communication with the duplex Avionics bus through the Bus Interface Unit (BIFU) as appropriate subsidiary tasks. Alternatively, a distributed executive could be incorporated; although this would minimise the number of processors it would impose additional requirements for processing and storage on those remaining and might lead to a nett penalty.

A dual duplex Utility bus arrangement is used to increase integrity through isolation, preserved by fibre-optic links to the BIFU. These links could also be used for inter-processor communication at executive level which would increase resilience to failures in both the control system and Utility components by facilitating interchange of status information and rescheduling of tasks.

Individual sub-system tasks such as fuel, air system control or brakes are assigned to groups of local terminals according to the level of redundancy required. In the event of local failures the tasks, including the sub-system executives, are re-assigned to a different terminal. In the event that this results in excessive loading on the remaining terminals the tasks would be executed on a priority basis, resulting in a gradual degradation of system performance and response time. This is preferable to abrupt failure, particularly in permitting time for corrective pilot action, for example changes of flight regime or aircraft configuration. Since many Utility sub-systems are only required in particular areas of the flight regime they lend themselves to this approach. Rescheduling of tasks could be controlled by the system executive processors, but in the event of failure here a task rescheduling protocol could be activated based on a distributed executive in local processors. This would also require a local executive to take over as bus controller, perhaps actuated after cross monitoring detects a failure. Obviously a new approach to data transmission which did not require a bus controller and which lent itself to the protocols of a distributed system would be advantageous in simplifying local units and software, and improving system reliability.

It is unlikely to be acceptable for single failures to lead to loss of control over the relatively large number of loads connected to a terminal, thus the use of dual redundancy is indicated at processor, data terminal and bus level for both the left and right channels. A possible terminal architecture is shown in Figure 9. Although the number of processors is now increased their required capability has been reduced by distribution and it is probable that suitable single-chip devices will be available in the time-scale of need. Since the mass and complexity of the local units has been shown to be dominated by the large number of interface circuits for the Utility components the practical penalties of the microprocessors will be small. To reduce the number of separate boxes to be installed the system executive processors, shown as stand alone units in Figure 7, could be integrated with local processors.

The methods used to identify failures at each level are critical design factors. Failures of Utility components and interfaces could be detected by their related local processor

using techniques such as loop tests, data reasonability checks, and comparison with effectively redundant data, for example comparison of gauged fuel quantity with calculated values based on integration of flow-rate. Processors could similarly test their associated data terminals. Failure detection for processors and data terminals involves comparison of lanes. Single faults can thus be detected with high confidence and BIT action initiated to determine the failed unit. The effectiveness of BIT arbitration could be improved by involving processors in other terminals to test the faulty terminal, since cross-monitoring in those terminals provides high confidence of their correct operation. The involvement of other terminals enables a voting procedure to be introduced without the penalties of increasing the redundancy level within individual terminals.

It can thus be seen that the quoted advantages of distributed processing such as high integrity, cost-effective use of processor resources, reduced vulnerability to battle damage and more gradual and easily contained degradation under fault conditions should be obtainable in the application to aircraft mechanical and electrical systems. Such systems have characteristics which lend themselves to the distributed approach, the major factors being:

- a) Mechanical and electrical components are widely distributed in the aircraft.
- b) Tasks are largely autonomous at the lower functional levels.
- c) The integrity and reliability requirements of different sub-systems vary widely so that a parallel redundant approach based on the most stringent requirements is inefficient.
- d) Most sub-systems can tolerate relatively long breaks in service enabling task rescheduling and reconfiguration to be achieved using resources normally performing lower level functions. The rapid response of synchronous parallel redundant systems is not required.
- e) The wide range of sub-system priorities and the capability to rank them according to flight regime is well suited to a distributed processing scheme in which efficient use is made of processing resources combined with a progressive reduction in system capability under failure conditions.

6. BENEFITS

The work to date has been based largely on the system shown in Figure 4. A comparison between this system and mechanical systems control on a current high technology aircraft (TORNADO) has highlighted the following quantifiable benefits:

- a) A mass reduction in the order of 50% of control system hardware and wiring which represents approximately 100 Kg in real terms.
- b) A volume reduction in the order of 30% of control hardware.
- c) A reduction in pilot workload of about 4:1 (based on analysis of an emergency situation and the modelling of a twin engine start routine).

Figure 10 shows these results compared with claims found in other published work. This figure also shows claims for improvements in reliability, maintainability and survivability. These last three parameters have not yet been addressed in the INCOMS work, but it is anticipated that future work will obtain similar results.

Further analysis of INCOMS will yield results that will enable us to establish reliability figures.

Advanced Health Monitoring and Maintenance recording techniques that are being considered in the Utility Systems should dramatically improve the Maintenance turn-round time.

Improved fault detection capability and the ability to reconfigure will improve survivability aspects.

7. CONCLUSIONS

Various techniques for the application of digital control to Utility Systems have been investigated in this paper. It has been shown that the preferred approach utilises a number of distributed processors and terminals that interface with the Utility components. Analysis performed to date shows that significant reductions in mass and pilot workload can be achieved.

Further work is required to refine the System's design and to assess other potential advantages of adopting INCOMS.

ACKNOWLEDGEMENTS

This work has been carried out with the support of Procurement Executive MOD. Where opinions are expressed they are those of the authors. The encouragement of the Director, Royal Aircraft Establishment, and by the Directors of British Aerospace, Warton Division, is acknowledged, as is the generous assistance of the authors' colleagues.

REFERENCES

- Collingbourne, L. R., 1980:
"Application of Digital Computer Control to Aircraft Electrical Systems"
TR80095 Royal Aircraft Establishment, Farnborough, UK.
- Collingbourne, L. R., 1981:
"Reliability and Integrity Aspects of Digitally Controlled Aircraft Utility Systems"
Report to be published, Royal Aircraft Establishment, Farnborough, UK.
- Durkin, H., 1977:
"Some Engineering Problems in the RAF"
AGARD-R-653.
- Kaye, A., Swetman R. E., 1979:
"Advanced Fuel Management System Study - Progress Report 1"
BAe Report TNAM 3343.
- Lancaster, P. A., Moxey C., 1980:
"Study of Systems Management for Future Military Aircraft - Volume 2"
BAe Report TNAM 3374, Volume 2.
- MASCOT Suppliers Association, 1980:
"The Official Handbook of MASCOT"
Royal Signals and Radar Establishment, Malvern, UK.
- Moir, I., Moxey, C., Lancaster, P. A., 1981:
"Command Response Data Transmission Applied to Mechanical Systems Management. Effect on the Crew System Interface"
AGARD 32nd Symposium of the Guidance and Control Panel.
- Moxey, C., Wilson, P., 1980:
"Advanced Fuel Management System Study - Progress Report 3"
BAe Report TNAM 3371.
- Ohlhaber, J. I:
"Aircraft Electrical Multiplex System"
IEEE Intercon Conference, March 1973.
- Ohlhaber, J. I:
"An Integrated Systems Approach to Helicopter System Design using Redundant Multiplexing Techniques"
5th European Rotorcraft and Powered Lift Aircraft Forum, September 1979.
- Ohlhaber, J. I:
"Integrated Multiplex for the Augusta A-129 Attack Helicopter"
6th European Rotorcraft and Powered Lift Aircraft Forum, September 1980.
- Rice, C. I., 1977:
"Avionic Solutions to Future Requirements"
Interavia News Letter No. 8888.
- Roth, S. P., Miller, R. J., Mihalow, J:
"Future Challenges in V/STOL Flight Propulsion Control Integration"
SAE Aerospace Congress, October 1980.
- Seabridge, A. G., 1980:
"Advanced Fuel Management System Study - Final Report"
BAe Report TNAM 3382.
- Seabridge, A. G., 1979:
"A Proposed System Management Centre for Future Military Aircraft"
IEE 3rd Int. Conference on Trends in On-Line Computer Control Systems.
- Smith, T. B., et al, 1978:
"A Fault-Tolerant Multiprocessor Architecture for Aircraft"
NASA CR3010.
- Ward, A. O., Forsyth, D. Y., 1980:
"SAFRA: Controlled Requirements Expression"
BAe Report TNAAS 484.
- Wilcock, G. W., 1978:
"Evaluation of a Data Transmission System Based on MIL-STD-1553A for Control of Electrical Power Distribution in Aircraft"
TR78137 Royal Aircraft Establishment, Farnborough, UK.

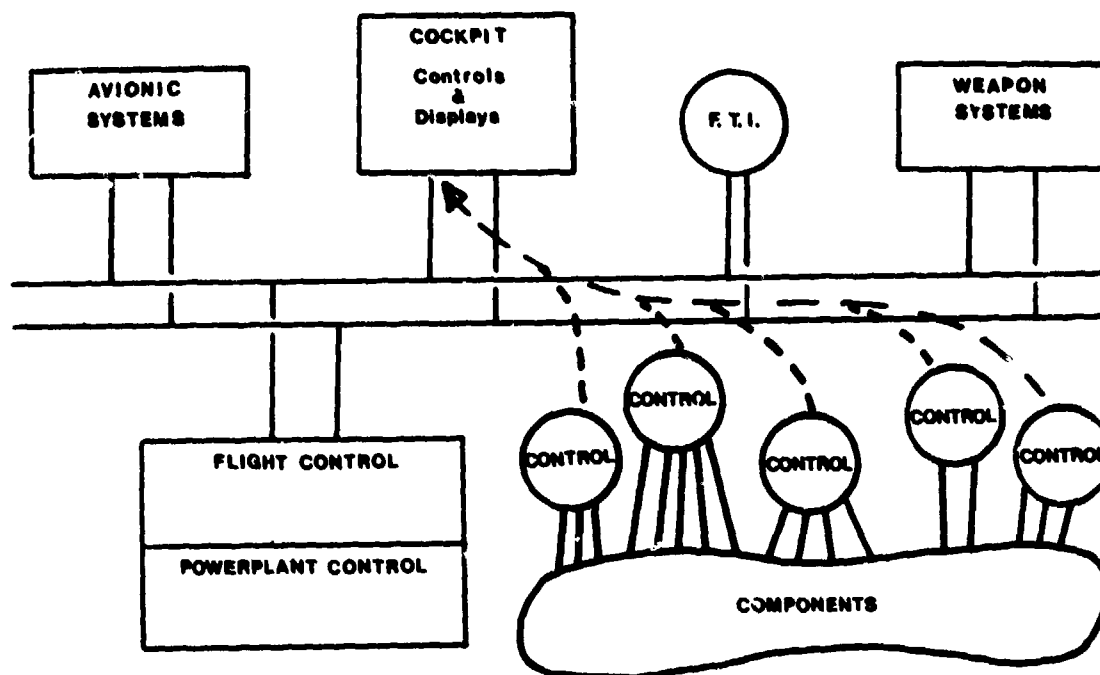


FIGURE 1 - UTILITY SYSTEMS MANAGEMENT WITH SEGREGATED CONTROL

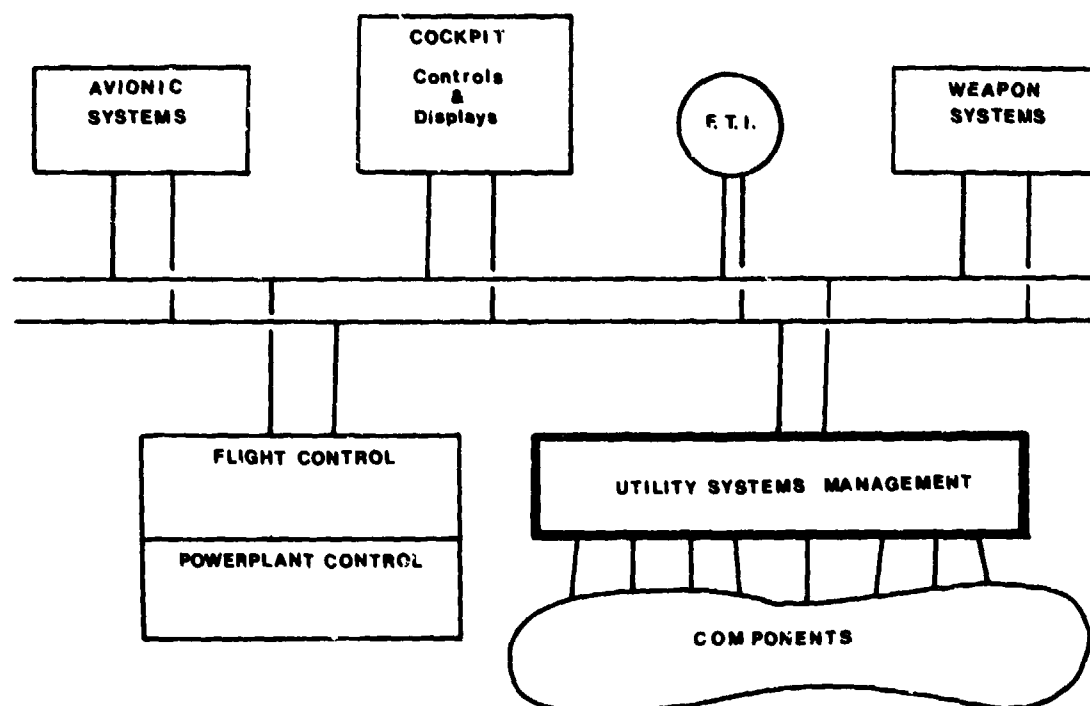


FIGURE 2 - UTILITY SYSTEMS MANAGEMENT WITH CENTRALISED CONTROL

KEY

- UTILITY SYSTEMS' COMPONENTS
- BUS INTERFACE UNIT
- INCOMS PROCESSORS

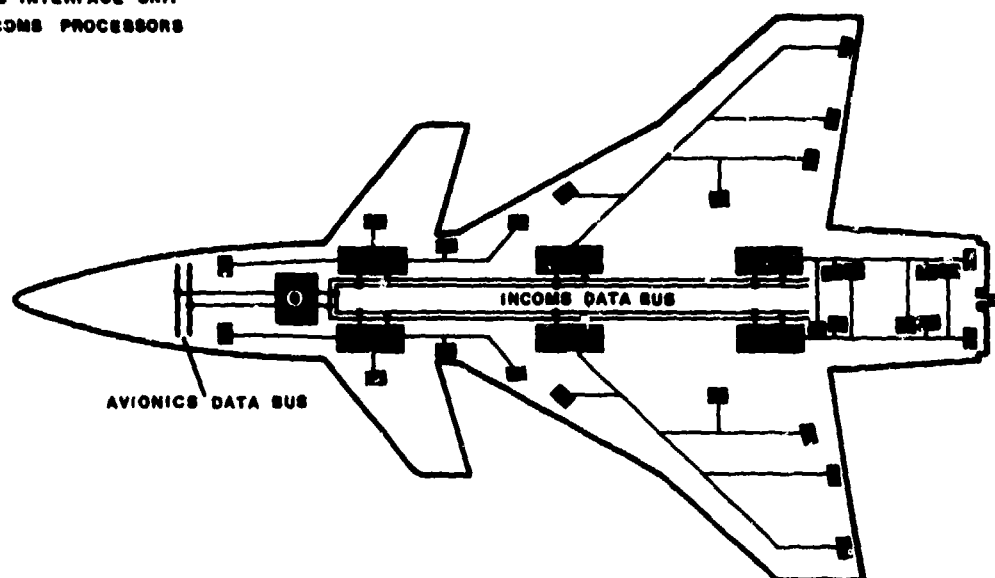


FIGURE 3 - TYPICAL AIRCRAFT: UTILITY SYSTEM LAYOUT

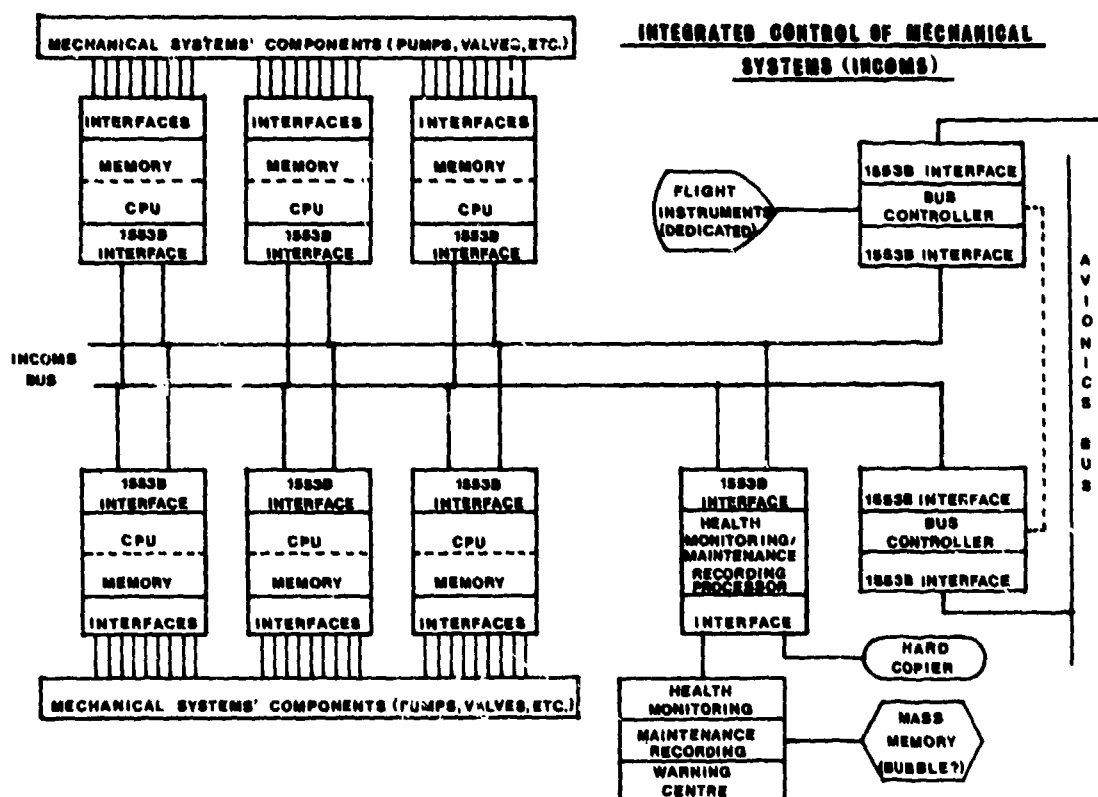


FIGURE 4 - THE INCOMS BLOCK DIAGRAM

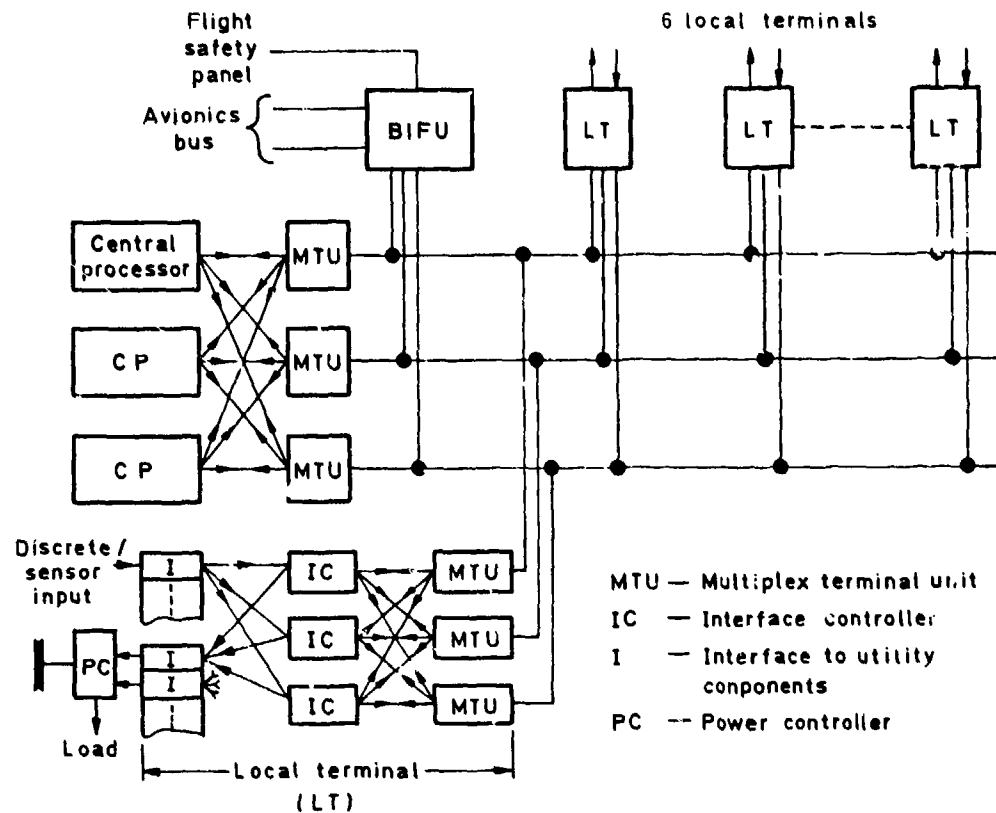


Fig 5 TMR control system

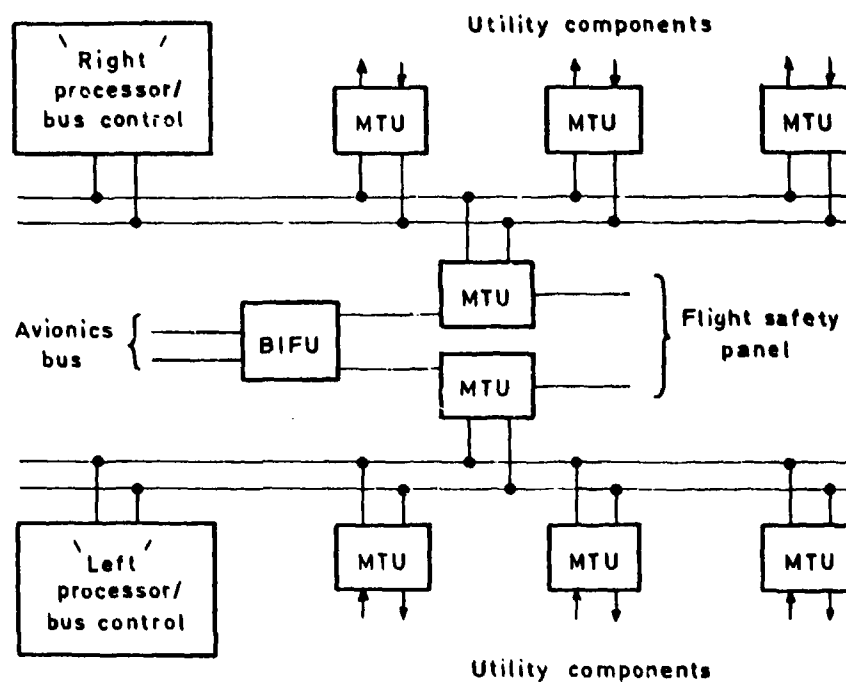


Fig 6 Centralized control system based on dual-duplex redundancy

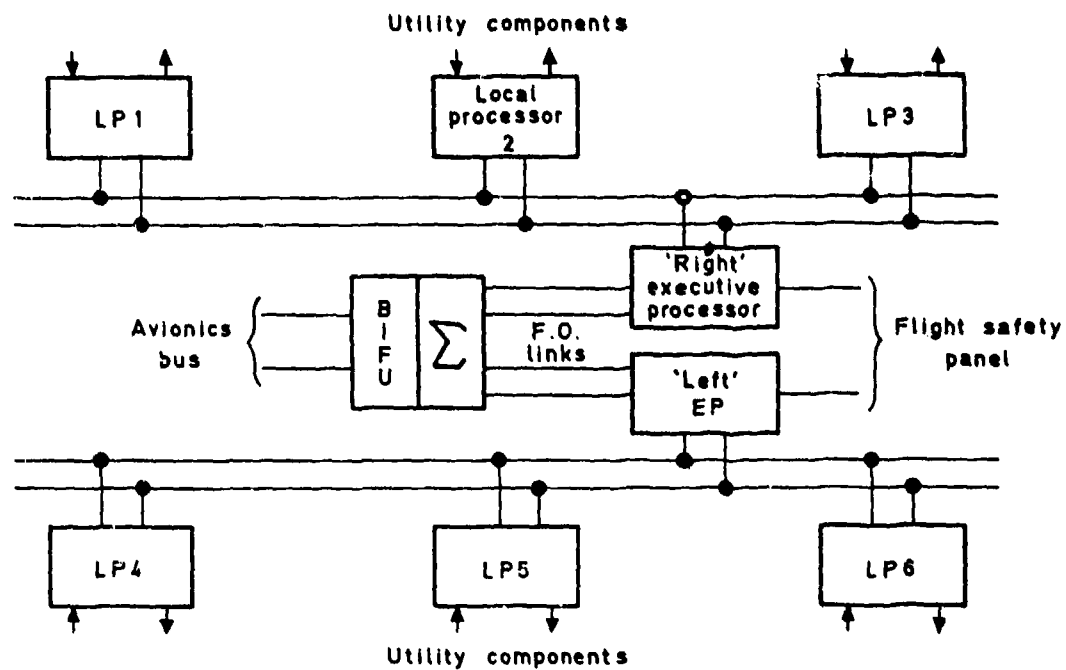


Fig 7 Distributed processor control system

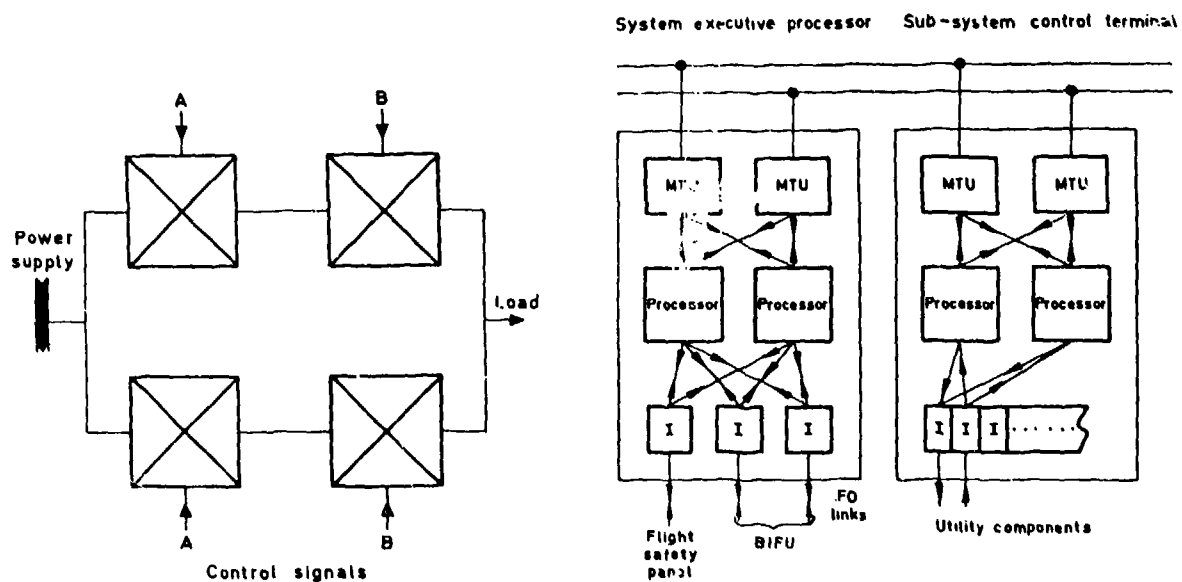


Fig 8 Redundant discrete output controller

Fig 9 Terminal architecture for a distributed processing system

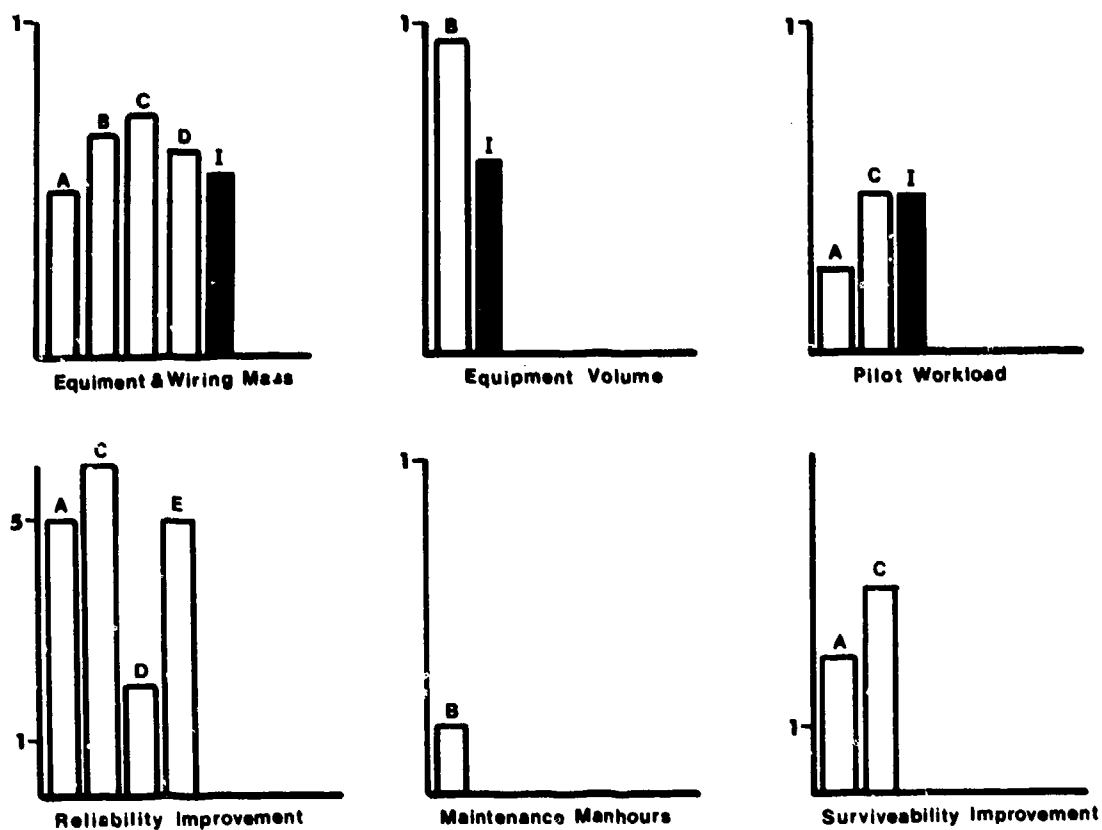


FIGURE 10 - MEASURES OF EFFECTIVENESS

KEY

A: Ohlhaber, J. I., 1980
 C: Ohlhaber, J. I., 1979
 E: Rice, C. I., 1977

B: Ohlhaber, J. I., 1973
 D: Roth, S. P., 1980

I is current situation

I denotes INCOMS results

ARCHITECTURE DU SYSTEME D'ARMES DU MIRAGE 2000

S. Croce-Spinelli
B. Vandecasteele
J.F. Ferreri
Avions Marcel Dassault-Breguet Aviation
78, Quai Carnot
92214 Saint Cloud
France

Résumé

L'architecture du Système d'Armes du MIRAGE 2000 représente une génération avancée de système numérique. Elle est décrite des points de vue:

- des équipements numériques
- de la répartition des logiciels entre ces équipements
- des liaisons numériques
- de la surveillance du système en vol.

On analyse les principes qui ont servi de base à la conception, les méthodes de développement et de test nécessaires.

On montre la flexibilité inhérente qui permet de s'adapter à différentes exigences opérationnelles et à différentes versions possibles.

ARCHITECTURE DU SYSTEME D'ARME DU MIRAGE 2000

1. INTRODUCTION

Le Système d'Armes du MIRAGE 2000 est pris comme exemple d'une génération nouvelle de systèmes numériques.

Il est intéressant de montrer les principes qui ont servi à sa conception et comment ces principes sont à la base de toute une famille de systèmes. Cette famille résulte de la flexibilité inhérente à ces principes, flexibilité qui permet ainsi de créer aisément des solutions de différentes "tailles" adaptées à différentes exigences opérationnelles sans nécessiter à chaque fois des investissements trop importants.

Ces principes d'architecture recouvrent à la fois des aspects "matériels", en particulier tout ce qui concerne les liaisons numériques entre les calculateurs, et des aspects "logiciels", en particulier la répartition des tâches entre les calculateurs et les interfaces correspondantes.

A cette génération d'architecture sont associées des méthodes de développement qu'il a fallu mettre au point pour prendre en compte la complexité et la variété des tâches que les systèmes correspondants sont en mesure de réaliser.

Dans cette famille de systèmes on peut classer des réalisations nombreuses déjà en production ou en cours de développement par notre Société.

On peut grossièrement classer ces réalisations par l'utilisation des liaisons numériques multiplexées. Le type "Digibus" n'est pas un même standard a été utilisé pour l'aéronautique militaire française depuis 1974 et qui est aussi utilisé pour de nombreux autres besoins (essentiellement militaires).

2. PRESENTATION GENERALE DU SYSTEME DU MIRAGE 2000

Le Système d'Armes du MIRAGE 2000 comporte essentiellement des équipements numériques programmables, dont la structure interne est adaptée aux besoins spécifiques de traitement: taille, mémoire, vitesse de traitement du processeur, spécialisation éventuelle du processeur, les organes de calculs organisés en "multiprocesseurs").

Sur le schéma fonctionnel général du système on peut distinguer:

- les principaux capteurs
- les visualisations et les commandes
- les calculateurs centraux qui réalisent notamment une gestion centralisée du système
- les organes d'interface
- le Digibus.

Les capteurs

On peut citer plus particulièrement:

- La centrale Inertie qui a été adoptée pour son utilité dans tous les rôles: avion, aussi bien la navigation, que l'air-air ou que l'atta. air-sol. La centrale comporte son propre calculateur.
- Le radar qui possède également des fonctions air-air comme des fonctions air-sol et navigation, et dont la "taille" résulte principalement des besoins en portée pour les interceptions air-air. Le radar fait la plus large part au traitement numérique du signal et possède également une unité arithmétique programmable assurant la gestion du fonctionnement du radar, les calculs et les échanges avec le reste du système.
- Le Système est également capable de recevoir d'autres capteurs de type électro-optique par exemple (en pods), qui sont reliés par le Digibus.
- Les contre-mesures passives (et actives) également numériques
- Les visualisations et les commandes
 - Les visualisations sont du type catodique. La tête haute possède un champ particulièrement important, pour permettre un confort maximum de pilotage dans toutes les phases de mission et pour permettre également les visées qui se font dans les différentes conduites de l'atta. air-sol, avec des hausses très différentes. On peut

aussi signaler que les angles d'incidence pratiqués par le MIRAGE 2000 sont très importants et que cela aussi entraîne la nécessité de champs élevés; par exemple, des hautes incidences sont utilisées pour obtenir des vitesses d'approche faibles. La tête basse, trichrome comporte la possibilité de présenter simultanément des images au standard TV et une symbolologie cavalière. Le générateur de symboles est numérique et comporte une unité arithmétique de type universel. Il existe également une visualisation cathodique spécialisée pour les contre-mesures.

Par ailleurs quelques instruments à indication analogique subsistent; ils sont parfois numériques.

• Les commandes comportent principalement:

- des commandes "temps réel" situées sur la manette et sur le manche de pilotage; elles permettent toutes les actions nécessaires dans les phases critiques des missions.
- un poste de sélection des armes et des modes entièrement piloté par logiciel.
- d'autres postes de commande, numériques, tels que le poste de commande de navigation.

- Les Calculateurs centraux

Ces deux calculateurs se partagent de nombreuses tâches de calculs associées aux missions de l'avion, en liaison avec tous les autres équipements numériques. Ils assurent notamment la gestion centralisée du système comme on le verra plus loin. En redondance, ils gèrent les échanges numériques sur le Digibus.

- Les Organes d'interface

On peut ranger dans cette catégorie les circuits armement et les interfaces avec les missiles.

On peut y ranger le pilote automatique qui assure la liaison entre les capteurs du système et les commandes de vol électriques pour réaliser les modes de base et les modes supérieurs de pilotage automatique.

Rentre également dans cette catégorie le boîtier de compatibilité contre-mesures.

- Le Digibus

On remarque que tous ces équipements numériques qui sont représentés sur le Synoptique sont reliés entre eux par la liaison standard de type "Digibus", dont la gestion est assurée, comme on vient de le dire, en mode normal par l'un des deux calculateurs centraux, en mode secours par l'autre.

A ce système entièrement numérique, il faut ajouter des moyens de radiocommunication, de radionavigation et d'identification. D'ailleurs certains de ces équipements font un large appel aux techniques numériques (IFF, TACAN...).

3. PHILOSOPHIE DE REPARTITION DES TACHES DE CALCUL

Lorsque l'on est passé progressivement de systèmes hybrides qui ne comportaient au départ qu'un seul calculateur numérique, aux systèmes de cette génération qui en comportent un grand nombre, il a fallu réfléchir à la manière de répartir les calculs, répartition qui fixe corrélativement les interfaces et les échanges entre les calculateurs.

Un certain nombre de critères apparaissent clairement pour ces localisations; ce sont les critères techniques liés aux caractéristiques des calculateurs (volumes mémoires et vitesse de calcul; et des liaisons (débit maximum d'échange sur le digibus). Il faut se rappeler en effet qu'on a affaire à des systèmes "temps réel" dans lesquels les contraintes temporelles sont essentielles (cadences d'échantillonnage ou de calcul, synchronisation et datation, pilotabilité...).

On n'insistera pas plus sur ces critères qui conditionnent au premier chef la faisabilité de toutes les solutions envisageables.

Par contre il y a d'autres critères qui sont moins évidents: critères humains, industriels, logistiques..., et pour lesquels nous pouvons donner quelques exemples.

Nous pouvons distinguer plusieurs catégories de calculs à l'intérieur du système:

a) Les fonctions "autonomes". Ce sont des fonctions spécialisées en rapport direct avec une technique particulière. Nous serons encore plus clairs en donnant trois exemples:

- la fonction "capteur" du radar qui consiste à extraire les informations de position, vitesse, accélération...des cibles.
- la fonction "capteur" de la centrale inertielle qui permet de mesurer la position, les attitudes, le cap...de l'avion.
- la fonction "tracé de symboles" du générateur de symboles de la tête haute et de la tête basse.

On peut noter que ces fonctions sont assez bien indépendantes les unes des autres d'où la dénomination d'autonomes (pour certaines fonctions on parle aussi de fonctions autonomes "assistées").

Ces fonctions peuvent se mettre au point séparément; elles nécessitent même souvent une mise au point préalable avant d'être intégrées dans un système d'armes complet à cause des problèmes techniques et technologiques qui y sont associés.

Ces fonctions font appel à des équipes de spécialistes chez des fabricants qui ont accumulé une expérience dans ce domaine d'équipements et dans leur maintenance.

Dans la technologie numérique actuelle qui met à disposition de tous les composants de base faciles à utiliser (encombrement et consommation faibles, outils logiciels de base développés...), ces fonctions doivent être incorporées directement dans les équipements correspondants (radar, centrale inertielle...) alors que dans le passé on a pu imaginer de les centraliser dans un seul calculateur.

b) Les fonctions "intégrées"

Un système ne s'identifie pas à la somme des fonctions autonomes que l'on vient de décrire. Il faut y ajouter un certain nombre de fonctions de coopération et de synthèse que l'on a appelées "intégrées". Elles permettent de créer les guidages de navigation et les conduites de tir au sens large.

Donnons quelques exemples:

- Calculs des lois de navigation pour l'attaque des cibles aériennes qui utilisent les données de tous les capteurs, à commencer par le radar bien sûr, qui sont fonction de l'arme sélectionnée et peuvent éventuellement évoluer avec l'expérience opérationnelle;
- Calculs des domaines de tir des divers missiles air-air;
- Calculs de tir canon air-air, par exemple calcul de la ligne de traceurs ou calculs à prédiction;
- Calculs de balistique pour les armes air-sol conventionnelles;
- Gestion des séquences de tirs en salve de bombes;
- etc.

Il faut ajouter à cette liste les fonctions fondamentales de gestion et de surveillance du système sur lesquelles nous reviendrons plus en détail au paragraphe suivant. Elles représentent un volume très important de mémoires programme (ce sont essentiellement des logiques). Elles dépendent très étroitement des missions et des besoins directs des équipages; elles ont en effet pour but d'assister les opérateurs humains dans l'utilisation d'un système complexe présentant un nombre extrêmement important d'états que l'homme ne peut gérer seul en temps réel.

Ces fonctions ne peuvent être attribuées a priori à un équipement donné. En dehors des critères techniques déjà évoqués on fait alors intervenir d'autres critères.

- Lorsque des fonctions dépendent étroitement de l'équipage on cherche à les rassembler dans les calculateurs centraux. Ce sont en effet des fonctions dont la mise au point se prolonge beaucoup et qui n'ont pu commencer qu'après l'intégration de tous les équipements composant le système (alors que la plupart d'entre eux ont fait l'objet d'une mise au point individuelle préalable.)
- Lorsqu'une fonction est étroitement liée aux caractéristiques d'un capteur on peut avoir intérêt par contre à faire appel à la même équipe de spécialistes et à localiser les calculs dans l'équipement. Cela a été le cas sur le MIRAGE 2000 pour les lois de navigation air-air qui sont dans le radar.
- Des considérations de probabilité de réussite de mission sont parfois prioritaires. Ainsi certains calculs de tir canon liés à des fonctions d'autodéfense sont localisés dans le générateur de symboles, de façon à mettre en oeuvre le minimum d'équipements nécessaires.

De la même manière on évite que tous les modes d'autodéfense soient dans le même calculateur (Canon et Magic) de façon à ce que de simples pannes ne suppriment pas tous les modes, sans pour autant nécessiter des redondances complètes.

La mise au point des fonctions citées présente d'ailleurs en général une assez bonne indépendance vis à vis des autres.

En résumé les critères pris en compte pour la répartition sont:

- la facilité de mise au point
- la souplesse d'évolution
- la compétence particulière de certaines équipes de spécialistes
- les critères logistiques: fiabilité des chaînes, maintenabilité...

Ils doivent être pesés cas par cas. D'une manière générale on a tendance à regrouper les fonctions dans les calculateurs centraux d'autant plus qu'elles sont intégrées, c'est-à-dire qu'elles associent un plus grand nombre d'informations élaborées dans des fonctions autonomes, et qu'elles sont plus proches des procédures opérationnelles et de l'utilisation par l'équipage. En particulier on y a mis toutes les fonctions de gestion d'ensemble du système qui nécessitent un volume important de logiques. La gestion du digibus avec son architecture redondante, est un cas particulier.

Il est certain que la situation est le résultat des possibilités technologiques et des moyens de développement de logiciel disponibles.

Elle est aussi caractéristique d'une certaine "taille" de système. Sont actuellement en développement, en parallèle sur le MIRAGE 2000, des systèmes plus simples - possédant une moins grande variété d'armes et de modes et conçus autour du calculateur de la centrale à inertie - et des systèmes plus complexes - possédant une architecture hiérarchisée. Tous utilisent cependant des calculateurs de puissance équivalente et des liaisons de type digibus.

4. LES PRINCIPALES FONCTIONS INTEGREES ET CENTRALISEES

Pour illustrer ce qui précède on peut revenir avec un peu plus de détails sur ce que l'on a appelé la gestion d'ensemble du système et plus particulièrement sur deux aspects:

- la gestion des commandes, des visualisations et des modes
- la surveillance permanente du système et la maintenance intégrée

Les logiciels correspondants sont regroupés dans les calculateurs centraux.

a) Gestion des commandes, des visualisations et des modes

Le but est d'apporter une assistance aussi grande que possible au pilote. On essaye ainsi que le pilote, en face d'un objectif ou d'un but de mission donné, n'ait si possible à décider que de l'arme à utiliser et que le système se charge des autres sélections: mise en oeuvre des équipements et des logiciels et mise en oeuvre des visualisations. On lui laisse seulement le choix des options, lorsqu'il y en a. L'accès aux modes doit être d'autant plus simple et immédiat qu'il s'agit de situations opérationnelles critiques (telles que l'autodéfense).

Sur la ligne inférieure du poste de sélection d'armes et de modes apparaissent les armes effectivement emportées par l'avion. Lorsqu'on sélectionne une arme (par défaut on est en mode navigation) apparaissent sur la ligne supérieure les seules options disponibles pour cette arme. Encore essaye-t-on de présélectionner les options les plus probables (que le pilote n'a à modifier qu'éventuellement).

Ces sélections entraînent automatiquement la mise en oeuvre des fonctions adéquates de tous les équipements (fonctions matérielles et logicielles). En particulier les réticules et symboles qui doivent apparaître sont entièrement déterminés (aux options près) par ces opérations.

Une commande reste cependant prioritaire par rapport à ce poste de sélection: c'est un bouton à trois positions situé sur la manette des gaz et accessible à tout instant. Ces trois positions correspondent à:

- Magic
- Canon
- Poste de sélection des armes et des modes

de sorte qu'on peut passer instantanément dans un mode d'autodéfense. (On change d'arme si on a déjà sélectionné un mode air-air).

Certaines commandes "temps réel" situées sur la manette ou le manche, sont utilisées différemment pour différents modes ou armes. Par exemple il existe une commande multiple qui pilote:

- en air-air: le décrochage radar et les différents modes d'accrochage automatique (axe, viseur, vertical)
- en air-sol: désignation de la cible, passage en navigation - passage en attaque (après présélection d'un mode d'attaque et des options correspondantes sur le poste de sélection on peut repasser en navigation tout en mémorisant le mode d'attaque).

En ce qui concerne la commande des visualisations; on peut signaler que des études de charge d'échange ont conduit à fixer l'interface de la manière suivante: la gestion centralisée adresse une liste des réticules à présenter à chaque instant. Par ailleurs le générateur des symboles reçoit (généralement à cadence beaucoup plus élevée) les variables pour les réticules mobiles; ces variables peuvent être soit des grandeurs physiques de base d'usage général (roulis, tangage...), soit directement le jeu de valeurs nécessaires au positionnement en axes viseur (exemple: un réticule de visée élaboré par la conduite de tir air-sol). Ainsi le générateur de symbole possède-t-il une bibliothèque de tous les réticules ou symboles, la gestion de leur présentation effective étant faite à chaque instant par un logiciel intégré et centralisé.

Il faut noter également que cette gestion fait intervenir l'état réel des équipements et de leurs fonctions. Il faut en effet ne présenter que les réticules valides et utilisables par le pilote. Cette gestion se fait au besoin par groupes de réticules qui sont nécessaires ensemble pour un même mode. En cas de panne on présente les consignes positives sur les modes qui restent disponibles, par exemple: passez en hausse manuelle, reportez-vous à la planche de bord (instruments secours)

Enfin pour chaque mode une trame différente peut être nécessaire pour les échanges sur le digibus. Le choix de cette trame fait également partie de la gestion centralisée.

On peut en résumé schématiser la gestion centralisée des commandes, des visualisations et des modes, comme un module logique dont les entrées sont:

- l'état des équipements
- les commandes actionnées par le pilote

et les sorties sont:

- la liste des visualisations
- les ordres de mise en oeuvre des fonctions matérielles et logicielles des équipements
- l'affectation des commandes en fonction du mode, la gestion des affichages sur le poste de sélection.
- le choix de la trame d'échange digibus.

b) Surveillance du système et maintenance intégrée

La surveillance permanente du système est basée sur l'utilisation des autotests permanents (ou cycliques) qui existent dans tous les équipements, et qui ont un taux d'efficacité élevé plus particulièrement dans les équipements numériques.

Le but premier de cette surveillance est de permettre une gestion complète des visualisations, des commandes et des modes pour décharger complètement le pilote de ce souci en ne lui laissant à disposition que les fonctions réellement opérationnelles.

Le résultat de ces autotests circule cycliquement sur le digibus. Il est utilisé par l'ensemble des équipements pour la constitution des validités de certaines chaînes et par la gestion centralisée comme on vient de le voir.

Une retombée de cette surveillance est une fonction de maintenance intégrée.

On peut en effet enregistrer les changements d'état des équipements pendant le vol dans les mémoires non volatiles du calculateur central. Ceci est utilisé au retour du vol pour déclencher les opérations de maintenance.

5. DIFFERENTES ARCHITECTURES CONCERNANT LES CALCULATEURS CENTRAUX ET LE DIGIBUS

Les choix ayant conduit au développement des différents matériels tels que calculateurs centraux et digibus nous permettent d'avoir à notre disposition un ensemble de modules s'adaptant parfaitement à différentes architectures de systèmes en fonction des besoins opérationnels.

Les modules de base principaux sont les suivants:

- Coupleur standard de bus permettant à un équipement d'être abonné sur un digibus (simple ou redondant)
- Coupleur procédure permettant de gérer un digibus
- Coupleur digibus CD84 assurant une gestion évoluée d'un digibus (simple ou redondant) ainsi que le mode abonné
- Fichiers, prises et câbles spécifiques assurant une excellente immunité aux parasites
- Coupleurs de sous-bus permettant d'étendre la procédure de base du digibus et en particulier le nombre d'abonnés.
- Répéteur de bus permettant d'interconnecter deux digibus en assurant un très bon découplage électrique entre eux (immunité au bruit).

La première architecture (développée) est conçue autour de deux digibus redondants gérés par un calculateur en mode normal et par un deuxième calculateur en mode secours en cas de panne du premier.

Le deuxième digibus n'est utilisé qu'en cas de panne du premier.

La deuxième architecture présentée (développée) diffère de la première par le fait que chaque équipement n'est connecté qu'à un seul digibus sauf en ce qui concerne les deux calculateurs gérants.

Les deux digibus sont gérés alternativement par le calculateur normal ou le calculateur secours en cas de panne du premier (débit équivalent 1Mbit/s).

Il y a par ailleurs un digibus avant et un digibus arrière sur l'avion considéré, et ceci pour des considérations de vulnérabilité.

La troisième architecture (développée) diffère de la deuxième par l'utilisation du calculateur secours. Celui-ci n'est plus dormant en mode normal mais travaille en abonné comme extension de volume mémoire et de puissance de calcul du premier calculateur.

En cas de panne du calculateur gérant en mode normal, le calculateur secours gère les deux digibus alternativement.

La quatrième architecture (en cours de développement) permet d'obtenir deux digibus indépendants travaillant en même temps et non plus alternativement (débit équivalent 2 Mbit/s).

Chaque calculateur, en mode normal, gère un bus et est en même temps abonné sur le deuxième; les transferts d'informations entre les deux digibus s'effectuent par l'intermédiaire de coupleurs COS intégrés dans les deux calculateurs.

En cas de panne d'un des deux calculateurs, le calculateur restant peut gérer les deux digibus alternativement comme dans l'architecture précédente.

L'architecture suivante présentée (système en projet) permet de généraliser le digibus à tous les points d'emport par extension de la capacité d'abonnés.

Ce système se distingue par:

- l'utilisation de coupleurs de sous-bus,
- la programmation des entrées:sorties analogiques vers les différents points d'emports,
- l'utilisation d'armements et d'emports sophistiqués qui indiquent au système qui ils sont et à quel point d'emport ils sont accrochés.
- l'utilisation d'une mémoire de masse permettant de charger les programmes nécessaires suivant la configuration des emports.
- des pylones avec une interface digitale et analogique standard et s'adaptant à différents bus (Digibus, Arinc 429 et MIL STD 1553-B) pour permettre l'interopérabilité des armes.

La dernière architecture présentée (développée) est caractérisée par une structure distribuée et non plus centralisée comme les précédentes.

6. METHODOLOGIE DE DEVELOPPEMENT DE LOGICIEL

D'une manière générale, la méthodologie apparaît comme une forme d'organisation technique, humaine et administrative du travail à réaliser permettant de:

- Définir, répartir et coordonner les activités du personnel.
- Prévoir et contrôler les délais, les coûts et à la qualité des travaux.

Les programmes assurant les différentes fonctions du système doivent être fiables et faciles à modifier.

Pour atteindre ces objectifs, la méthodologie doit reposer sur les principes suivants:

- Collaboration étroite entre l'avionneur et les fabricants du logiciel
- Procédures de test et de validation très poussées
- Décomposition des tâches par étapes parfaitement définies
- Emploi d'aides automatisées.

Le développement du logiciel comporte trois phases principales décomposées en étapes:

- Définition du logiciel:
 - . spécifications fonctionnelles globales
 - . spécifications détaillées des fonctions opérationnelles;
- Réalisation du logiciel:
 - . spécifications détaillées du logiciel
 - . analyse globale
 - . analyse détaillée
 - . développement des méthodes de test
 - . programmation et mise au point
 - . test d'ensemble
- Validation du système:
 - . essais au sol statiques et dynamiques
 - . essais en vol

La phase de définition du logiciel est placée sous la responsabilité de l'avionneur, avec la participation étroite des utilisateurs, des équipementiers et des fabricants de logiciel.

Elle consiste à établir deux types de documents:

- Les spécifications fonctionnelles globales du système dans lesquelles les fonctions sont décrites d'un point de vue opérationnel sans tenir compte du découpage de celles-ci entre les différents équipements.
- Les spécifications détaillées des fonctions opérationnelles qui spécifient, pour chaque fonction à réaliser, le découpage du logiciel entre les différents équipements et les tâches à effectuer par chacun d'eux.

. La phase de réalisation du logiciel est de la responsabilité de chaque fabricant de logiciel, c'est-à-dire, en général de chaque fabricant d'équipement.

Les trois premières étapes précisent toutes les informations nécessaires à l'écriture et à la mise au point des programmes: organigrammes, bilan mémoire, charges de calcul, découpage en modules, moyens de tests,...

Les deux étapes suivantes se déroulent simultanément:

- Développement des moyens de tests
- Programmation et mise au point en statique des différents modules.

La dernière étape permet de s'assurer que les programmes sont conformes aux spécifications. Les tests sont faits en dynamique sur une baie de validation de logiciel fournissant en temps réel des jeux de paramètres cohérents théoriques.

. La phase de validation du système est de nouveau de la responsabilité de l'avionneur. Il s'agit de tester les différentes fonctions dans un environnement opérationnel avec les véritables équipements et un câblage conforme à celui de l'avion.

Un banc d'intégration est utilisé dans un premier temps pour des essais statiques (essais d'équipements, vérification des câblages, mesures de précision...) puis des essais dynamiques à l'aide d'un ensemble de stimulation.

Cet ensemble de stimulation permet d'injecter au niveau des capteurs du système (centrale à inertie, radar, centrale aérodynamique,...) un jeu de paramètres cohérents en temps réel et préalablement enregistrés en vol ou sur des simulateurs. Ce système permet de "rejouer" un vol d'essai, une phase particulière de la mission, autant de fois qu'on le désire afin de faire des mesures ou des enregistrements particuliers.

C'est au cours de cette phase d'essais au sol qu'un certain nombre de modifications à apporter au logiciel vont apparaître, dues à des erreurs de programmation, des changements ou des précisions à apporter aux spécifications.

Enfin la phase d'essais en vol se termine par l'acceptation du logiciel qui sera implantée dans les équipements de série.

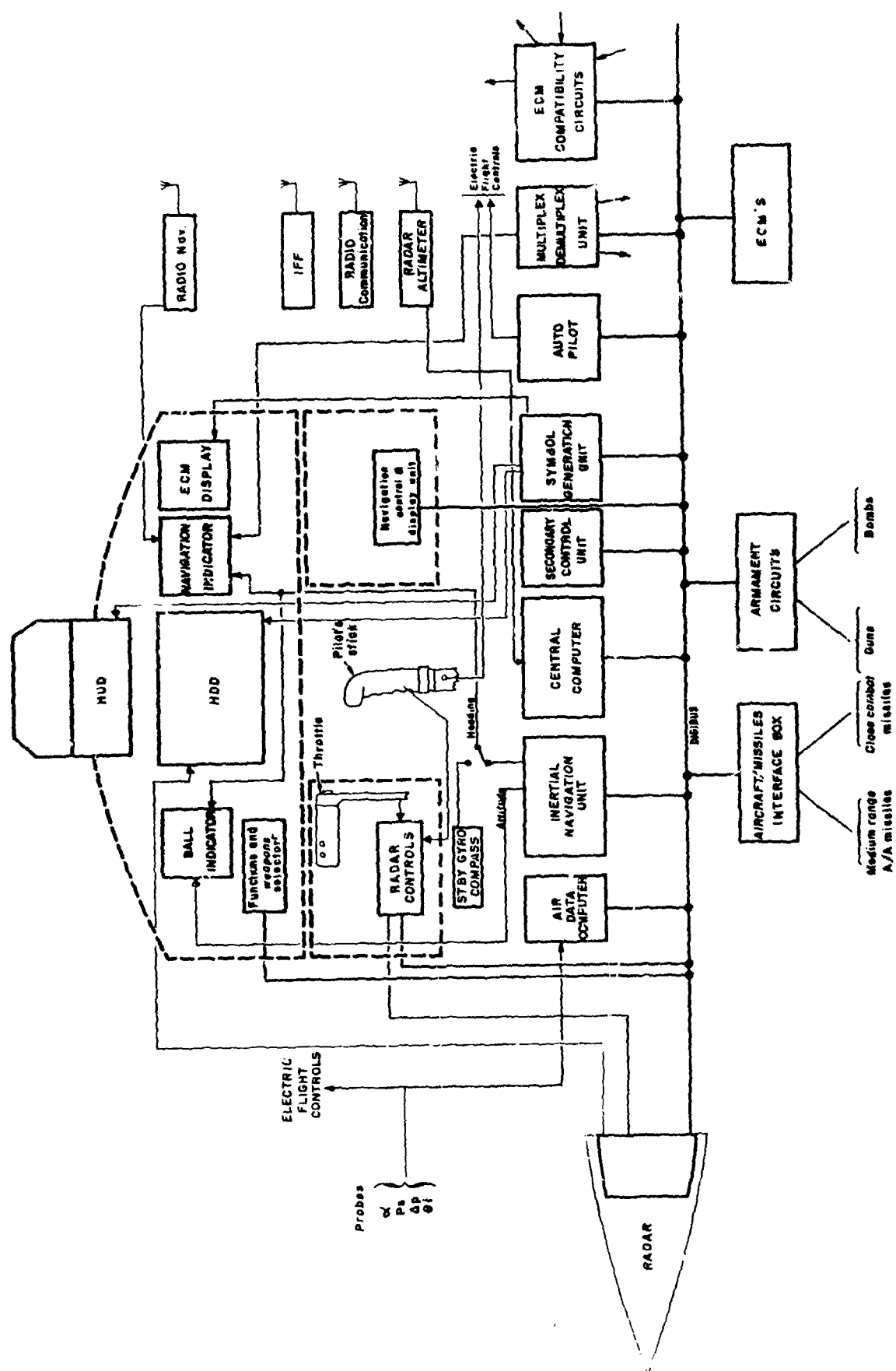
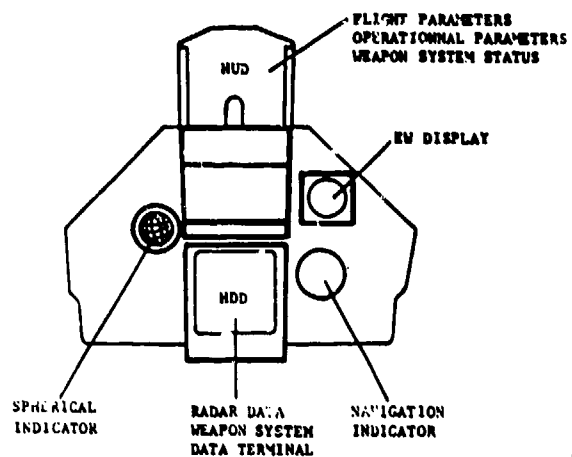
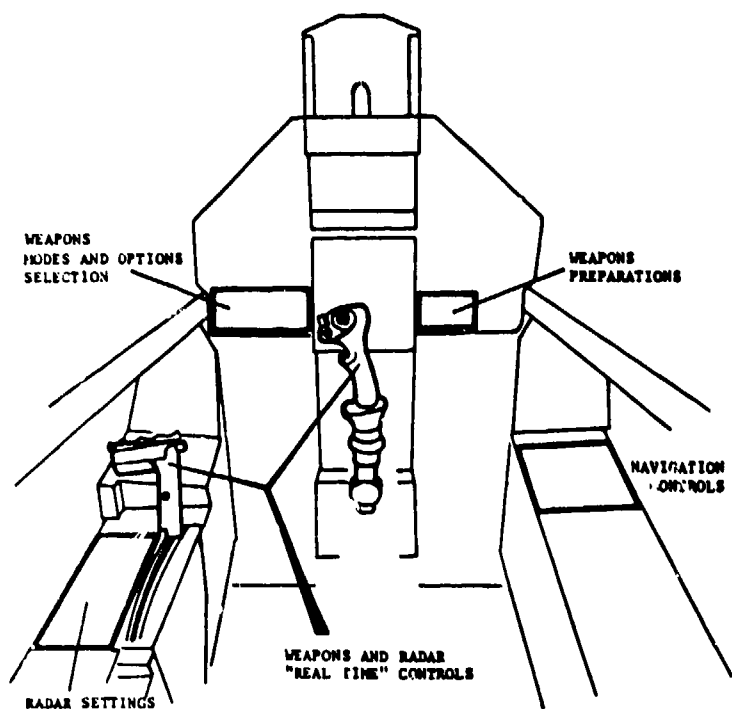


Fig.1 Mirage 2000 - nav/attack system



- . CRT HEAD UP } DISPLAYS
HEAD DOWN }
- . CENTRAL SITUATION
- . LARGE FIELD OF VIEW
 - . NAVIGATION
 - . GUNS AND ROCKETS
 - . LOW AND HIGH DRAG BOMBS
 - . APPROACH
- . PRESENTATION OF ALL NEEDED INFORMATION
IN A SINGLE PLACE
FOR EACH PHASE OF A MISSION

Fig.2 Displays



- . "REAL TIME" CONTROLS ON THE STICK
AND ON THE THROTTLE
- . CENTRAL MODES AND OPTIONS SELECTION
- . SOFTWARE AIDED SELECTION :
- ONLY POSSIBLE OPTIONS AVAILABLE AT ONE TIME

Fig.3 Controls

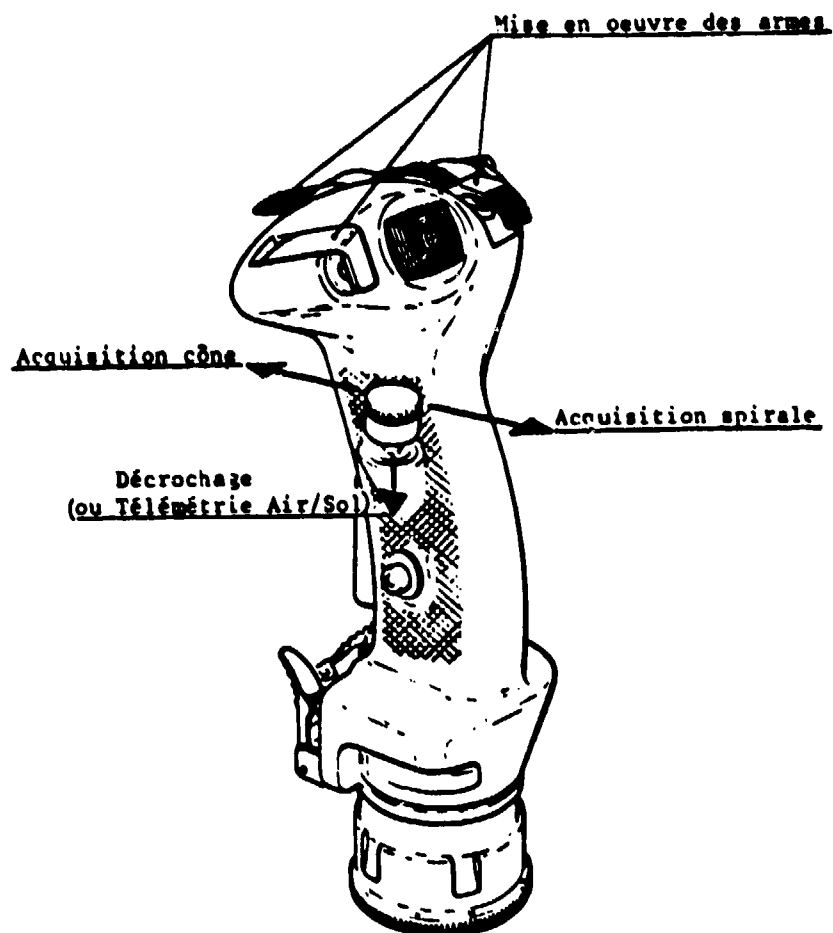
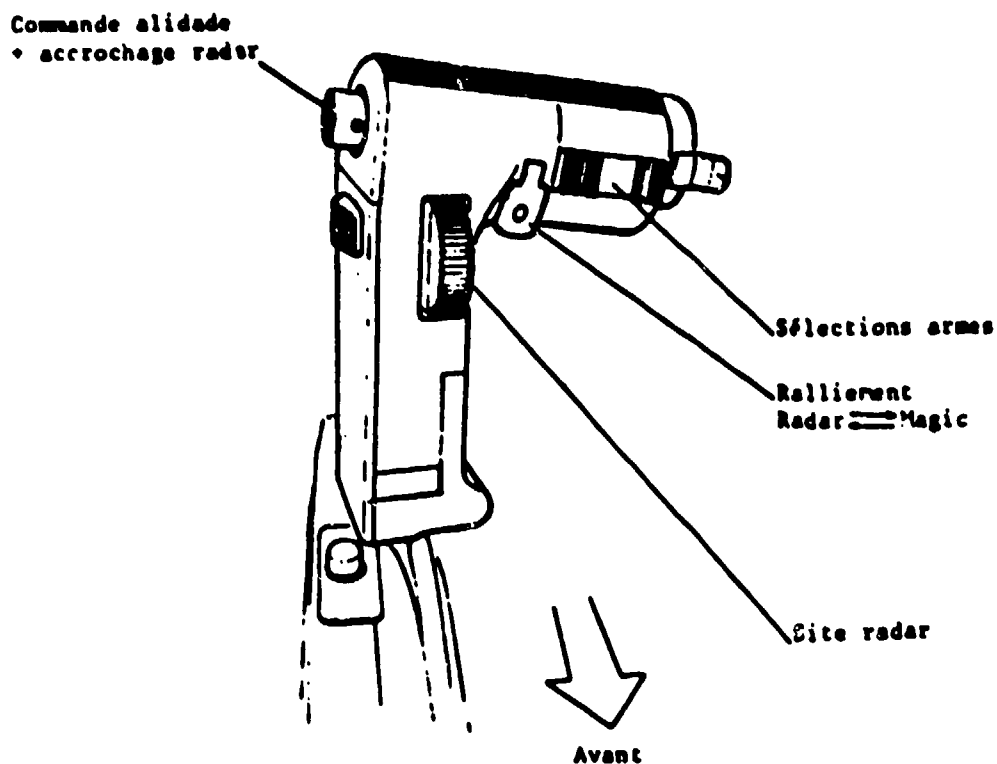


Fig.4 Mirage 2000 – poignée pilote



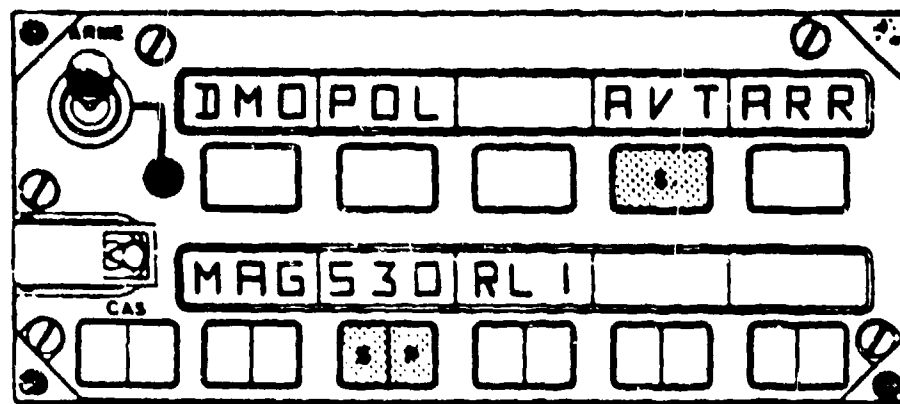


Fig.6 Poste de commande armement

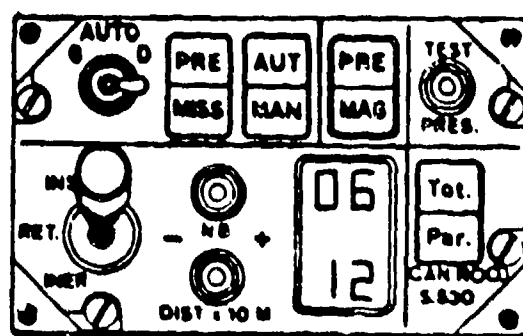


Fig.7 Poste de preparation armement

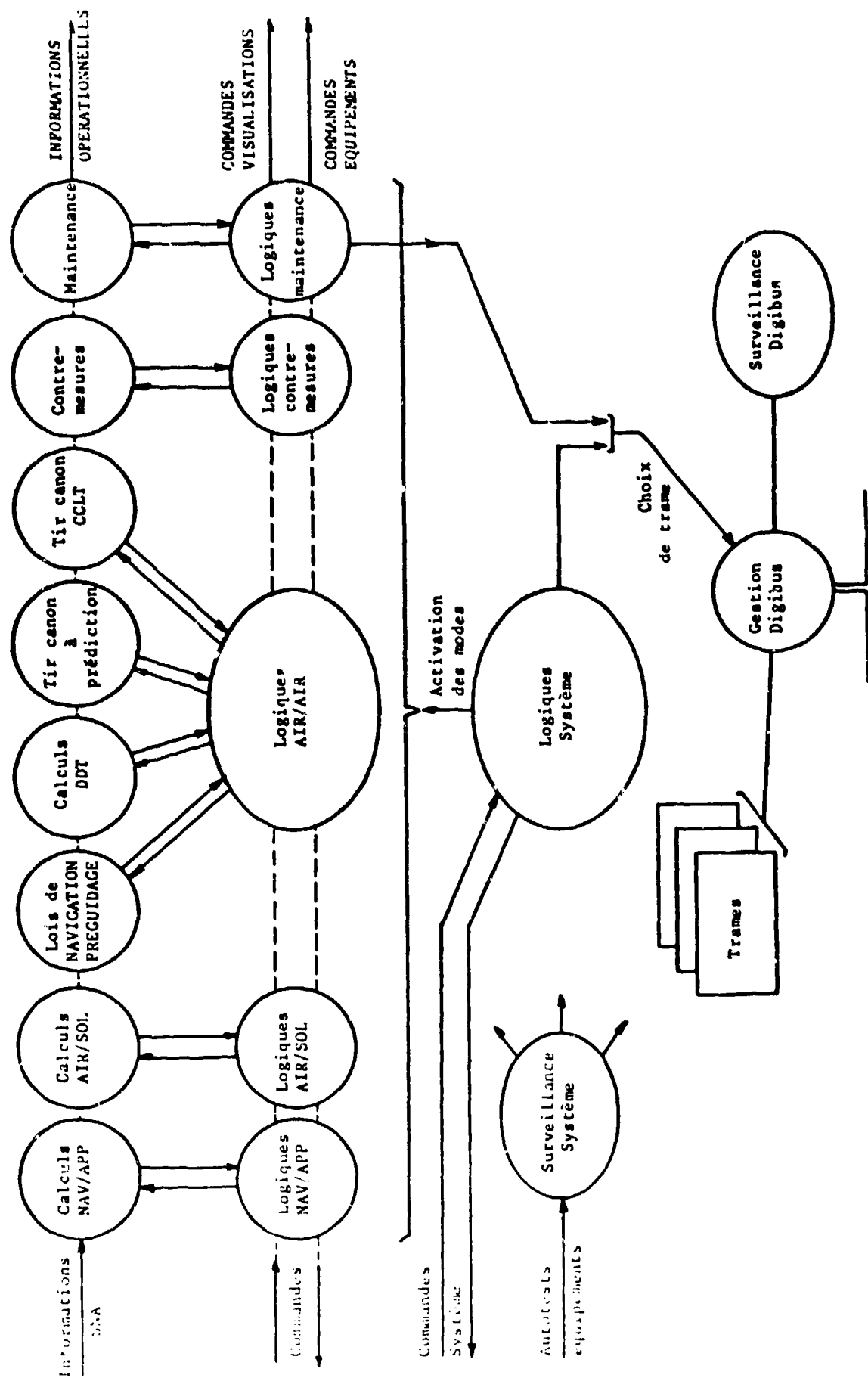


Fig.8 Architecture du logiciel

AND HAVE RECOMMENDED AND STUDIED FOR MANY YEARS THE USE OF DIGITAL TECHNIQUES.

- . 1961 : IDEA OF USE OF A DIGITAL COMPUTER FOR NAV/ATTACK COMPUTATIONS
- . 1967 : SUGGEST THE USE OF MULTIPLEXED DIGITAL BUSES
- . 1968 : DEVELOPMENT OF A MULTIPLEXED DIGITAL BUS AND STUDY OF PROTECTION AGAINST INTERFERENCES.
- . 1970 : STUDY WITH EMD OF GINA DIGIBUS
- . 1973 : DEVELOPMENT OF THE FIRST AIR TO GROUND ATTACK INTEGRATED SYSTEM USING AN UNIVERSAL DIGITAL COMPUTER, INU AND CRT HUD
- . 1974 : SUPER ETENDARD AIRCRAFT
- . 1974 : DEVELOPMENT WITH EMD OF GINA DIGIBUS
- . 1975 : DIGIBUS GINA HAS BEEN SELECTED FOR SUPER MIRAGE AND MIRAGE 2000 INTEGRATION TESTS
- . 1975 - 1980 : DIGIBUS GINA HAS BEEN OPERATED ON INTEGRATION BENCHES AND FLIGHT TESTS -

- . IT IS USED ON ALL MODERN AIRCRAFT :
 IN DEVELOPMENT : MIRAGE 2000
 ATLANTIC
 MIRAGE F1
 IN SERIAL PRODUCTION : MIRAGE F1

- . IT IS STANDARDIZED IN THE MILITARY A/C, SHIPS, MISSILES

Fig.9 History of Gina Digibus

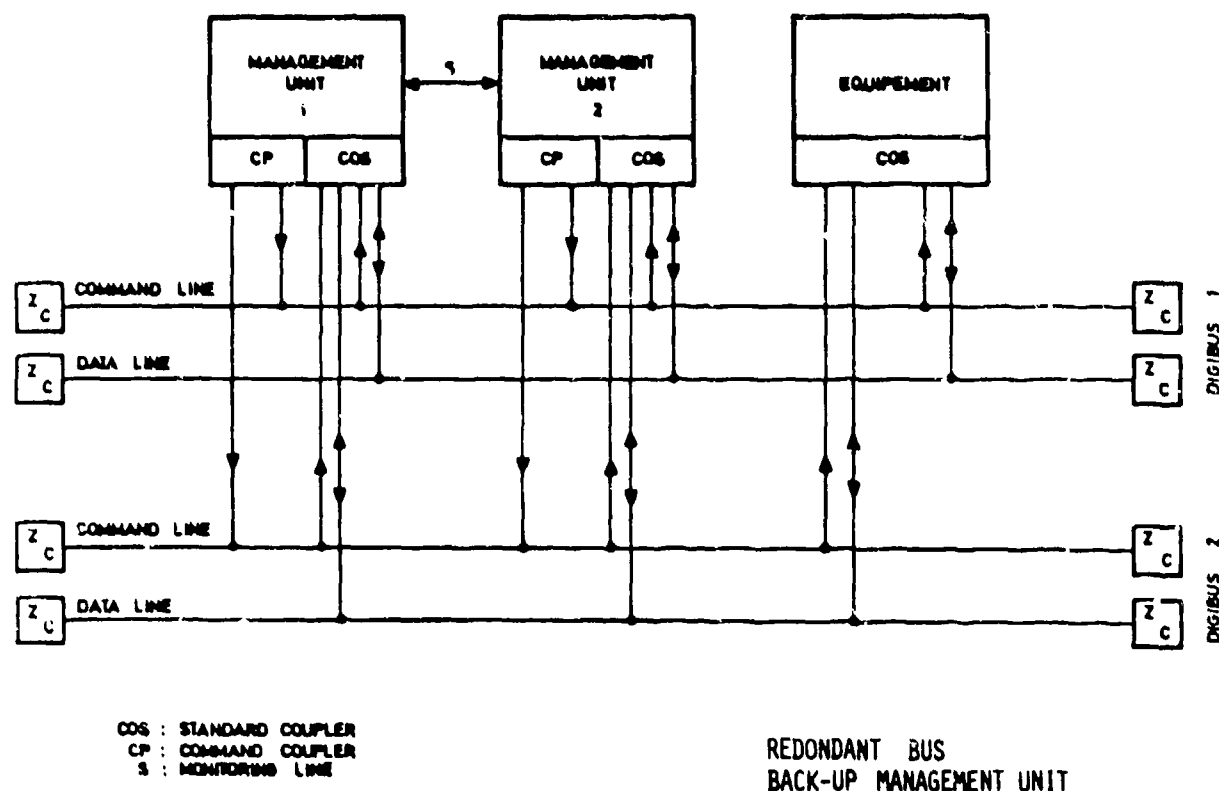


Fig.10 Architecture 1

FRONT AND REAR BUS FOR TACTICAL PROBLEMS.
 TWO MANAGEMENT UNITS : MAIN AND BACK-UP UNITS
 FLIP-FLOP MANAGEMENT OF THE TWO BUS
 EQUIVALENT : 1 Mbits/s

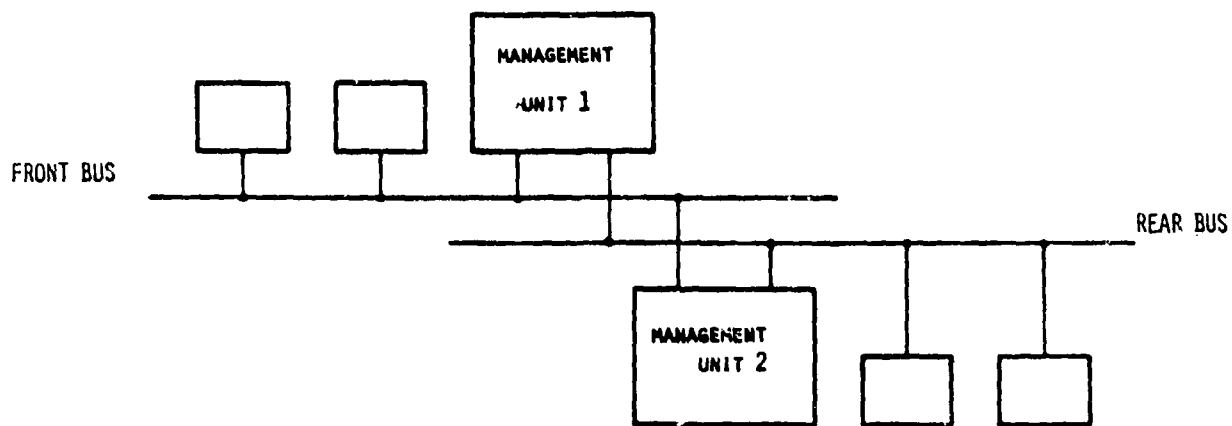


Fig.11 Architecture 2

BACK-UP MANAGEMENT UNIT PERFORMS ADDITIONAL COMPUTATIONS IN NORMAL MODE

BACK-UP MODE : (FAILURE OF ONE MANAGEMENT UNIT) = IDENTICAL TO NORMAL MODE BUT WITH FEWER TASKS

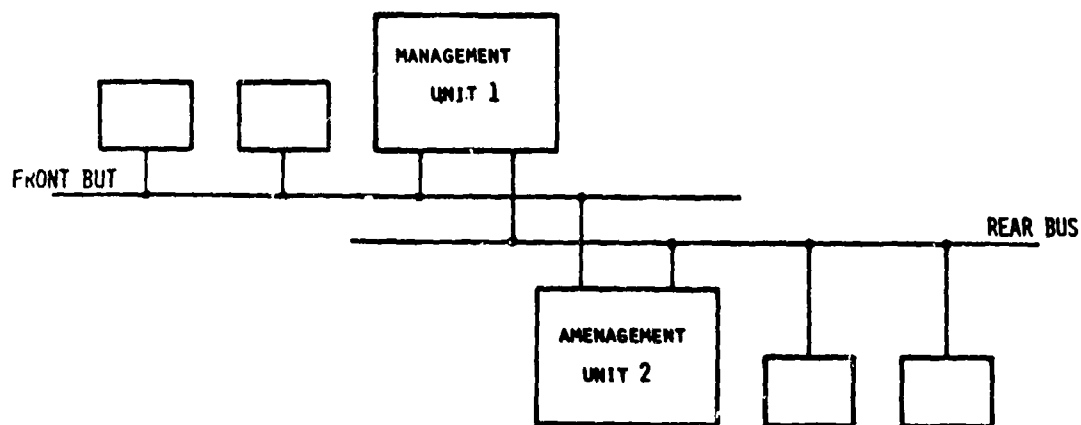


Fig.12 Architecture 3

TWO SEPARATE BUS EACH ONE CONTROLLED BY A COMPUTER : EQUIVALENT 2 Mb/its/s

EACH COMPUTER CONTROLS ITS BUS AND IS CONNECTED TO THE OTHER ONE

BACK-UP MODE : EACH COMPUTER CONTROLS THE TWO BUSES BY FLIP-FLOP

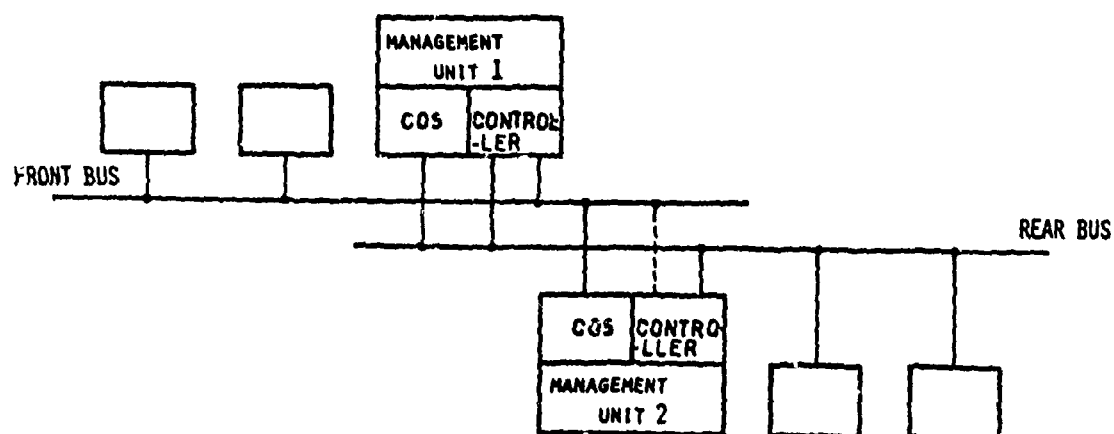
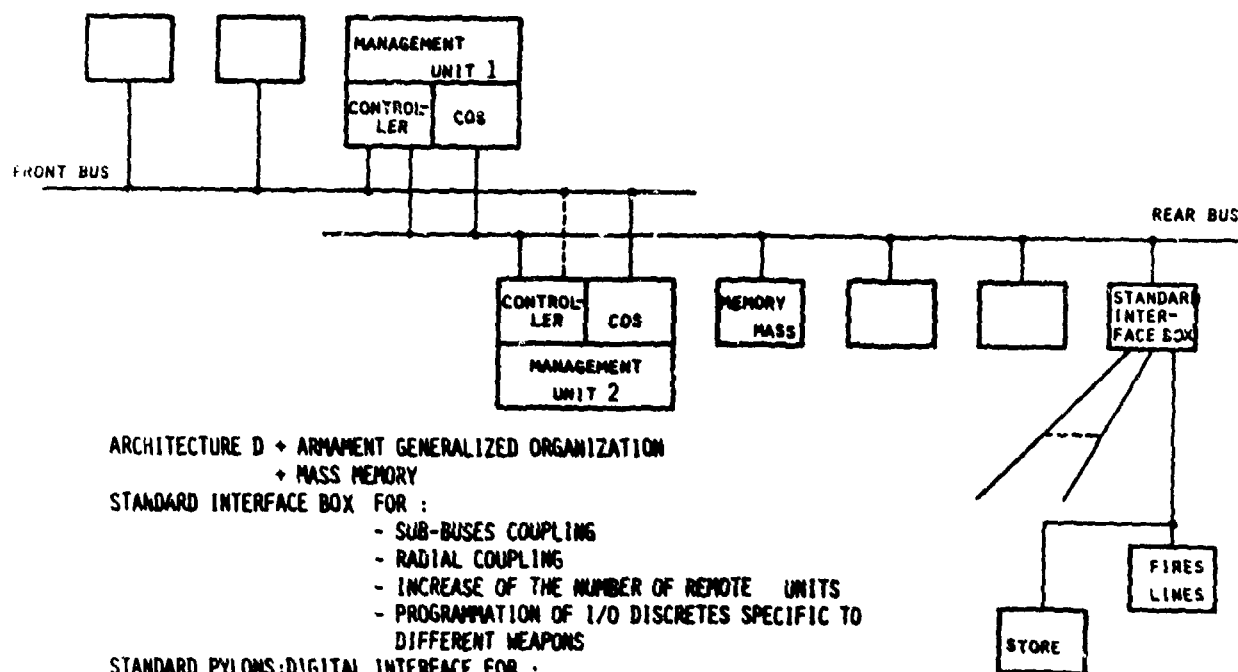


Fig.13 Architecture 4



ARCHITECTURE D + ARMAMENT GENERALIZED ORGANIZATION
+ MASS MEMORY

STANDARD INTERFACE BOX FOR :

- SUB-BUSES COUPLING
- RADIAL COUPLING
- INCREASE OF THE NUMBER OF REMOTE UNITS
- PROGRAMMATION OF I/O DISCRETES SPECIFIC TO DIFFERENT WEAPONS

STANDARD PYLONS: DIGITAL INTERFACE FOR :

- FIRE LINES
- DIFFERENT STANDARDS INTERFACE

Fig.14 Architecture 5

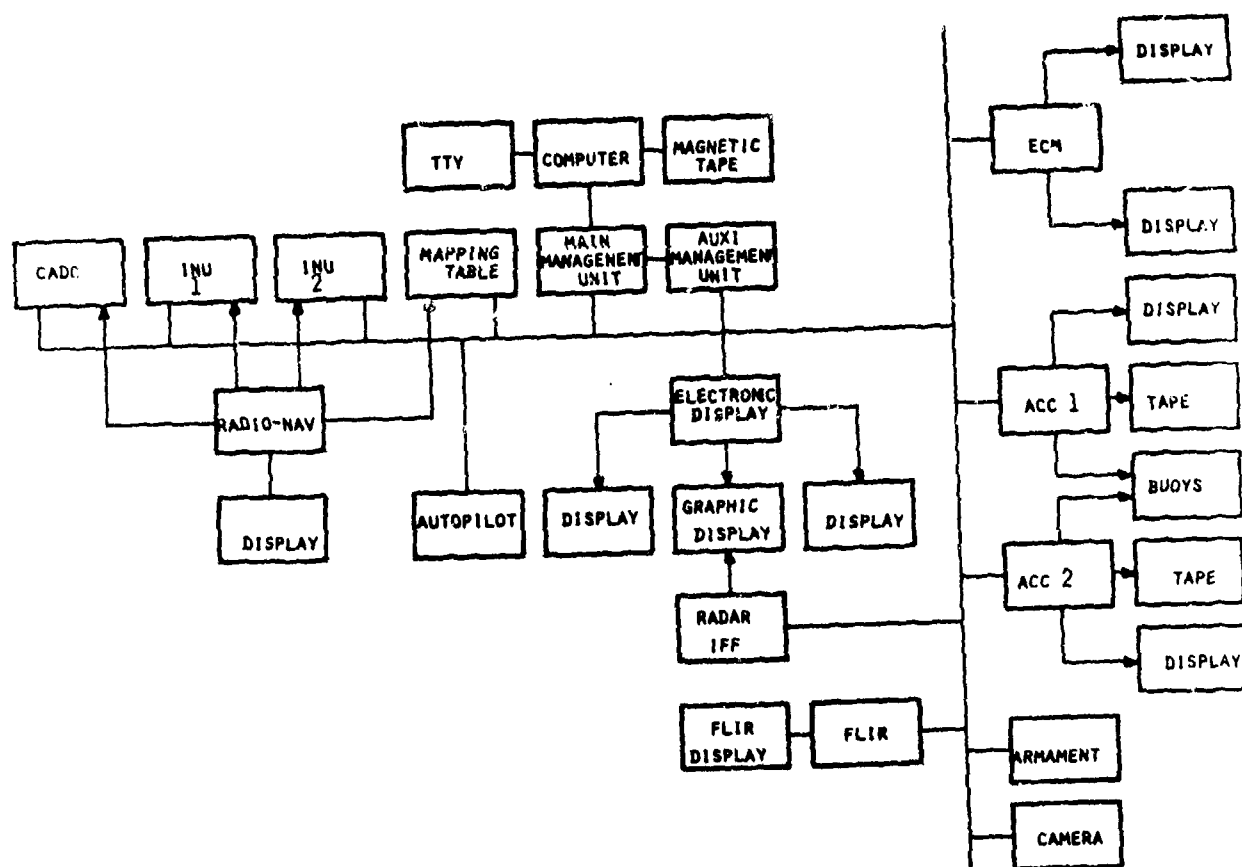


Fig.15 Architecture 6

F/A-18A TACTICAL AIRBORNE COMPUTATIONAL SUBSYSTEM

T. V. McTigue
Branch Chief
McDonnell Aircraft Company
McDonnell Douglas Corporation
Post Office Box 516
St. Louis, Missouri 63166 U.S.A

ABSTRACT

This paper presents a description of the Tactical Airborne Computational Subsystem used in the U.S. Navy/McDonnell Douglas F/A-18A Hornet Fighter/Attack Weapon System. It describes an airborne processing system of physically distributed computer resources interconnected through a multiplex communication network. Specifically, the paper describes the design, development, and integration of the Tactical Airborne Computational Subsystem for the U.S. Navy/McDonnell Douglas F/A-18A Hornet Weapon System. The F/A-18A Hornet tactical computer subsystem consists of two central Mission Computers and a number of distributed processors embedded in various sensor and display subsystems. This distributed processing system is interconnected by and communicates over a MIL-STD-1553A serial 1 MHz command/response multiplex network. The distributed processing system architecture is discussed and the rationale is presented for the partitioning of the computational tasks between the central Mission Computers and the distributed processors embedded in the sensor subsystems. The salient features of the central Mission Computer and the distributed processors are discussed along with a description of the functional operation of the interconnecting MIL-STD-1553A multiplex communications system. Finally, the development process for the Operational Flight Program (OFP) for the central Mission Computers is described including a discussion of the support facilities which were used for the software integration and validation.

1. INTRODUCTION

The purpose of the F/A-18A Hornet Weapon System is to deliver air-to-air and air-to-ground weapons on targets that must be detected, identified, acquired, tracked, and destroyed by the pilot using sophisticated sensors and weapons. In the course of an F/A-18A Hornet mission, millions of split-second computations and decisions must be made within the aircraft. The pilot, in addition to flying the aircraft, must constantly monitor the instruments and interpret the readings to ensure that the weapon system can accomplish its purpose. One-man operability was a prime goal in the design of the F/A-18A Hornet. Every decision and task that could be safely removed from the pilot was incorporated in a highly integrated computational subsystem. The operations within the subsystem are still at the pilot's command, but he is able to perform his primary tasks with confidence based on reliable, real-time operation of his computational subsystem. This subsystem consists of two mission computers and a number of distributed computers in various sensor and display equipments. The Operational Flight Program (OFP) for the Mission Computer was developed by McDonnell Douglas Corporation, St. Louis, Missouri, and is being flight-tested and qualified by McDonnell Douglas and Navy pilots at the Naval Air Test Center, Patuxent River, Maryland. The first U.S. Navy squadron was activated at the Naval Air Station in Lemoore, California, in February 1981, and the first production aircraft will be delivered in July 1981. The F/A-18A has also been selected by Canada and is under serious consideration by a number of other U.S. allies.

2. DATA PROCESSING SUBSYSTEM (DPS)

The DPS consists of two Mission Computers (MC) and several distributed computers in various sensor and display equipments. The Mission Computers, which employ the U.S. Navy AN/AYK-14 standard computer, integrate the overall operation of the avionics. The rationale for two Mission Computers was the same as for two engines. When they both are operational, they provide increased weapon system performance. When one is not operational, the other provides enough performance for self-defense and safe return.

The airborne computational requirements were classified into two major categories (Figure (1)):

- o Sensor-oriented computations
- o Mission-oriented computations

Sensor-oriented computations are defined to be those independent computations, such as sensor coordinate transformations, platform management, and signal processing, which are peculiar to a particular sensor or display. Mission-oriented computations, such as weapons launch calculations, are defined to be those computations directly related to performing the mission and dependent on the integration of information from several avionics subsystems. Table I shows typical examples of the two categories of computations.

TABLE I - COMPUTATIONAL CATEGORIES

<u>Sensor-Oriented</u>	<u>Mission-Oriented</u>
o Air Data Calculations	o Air-to-Air steering and launch zones for gun and missiles
o Radar Signal Processing	o Air-to-Ground steering and release for bombs, rockets, gun, and missiles
o Inertial Platform Management	o Selection of best available data from various sensors
o Display Symbol Generation	o Integrated display management

A system design technique was used on the F/A-18A Hornet that produced a set of Integration Block Diagrams which were used to partition the system requirements into specific tasks for each subsystem onboard the aircraft. The total airborne computational tasks were partitioned into mission-oriented tasks allocated to central mission computers and into sensor-oriented tasks performed in distributed processors in the sensor subsystems (Figure (2)). This relieved the central computers of those tasks which could be more effectively performed and managed in distributed and independent sensor processors. This approach offered functional modularity of the sensors, whereas system integration was provided by the Mission Computers. Hence, improved sensors and displays can be added later to the Avionics System, and present ones can be changed, with minimum impact on other equipment. Likewise, if the armament is altered for new or modified weapons as the mission of the aircraft is enlarged, such changes can be accommodated primarily through changes to the Mission Computer and Stores Management Set software.

A top-down software design approach is used, which partitions each computer program into software modules of manageable size based on functional groupings of computational tasks. The rationale for modular software is analogous to that for modular hardware. First, it permits each module to be independently developed, debugged, and tested in parallel with the other modules. Second, it allows changes to occur within a module without causing changes to be made in other modules, as long as the external modular interface remains the same. Analogous to the modularity and controlled interfaces in the hardware, new programming modules can be added and old ones deleted without impacting the whole program as long as the module interfacing rules are followed. Documentation and understanding of the total computer program is simplified, because each module can be described and understood as a separate entity.

2.1 Sensor-Oriented Processing

On-board the F/A-18A Hornet there are four major subsystem-embedded reprogrammable computers and a number of smaller subsystems with embedded microprocessors with Read-Only Memories (ROM). Table II summarizes the computer hardware for the major subsystems with reprogrammable computers. Table III presents the computer hardware information for the subsystems with ROM computers.

TABLE II - SENSOR-ORIENTED REPROGRAMMABLE COMPUTERS

COMPUTER	CPU	SPEED	MEMORY
INERTIAL NAV COMPUTER	2901	238 KOPS	16K CORE
RADAR DATA PROCESSOR	2901	700 KOPS	250K DISK/ 16K RAM
RADAR SIGNAL PROCESSOR	54S181	7100 KOPS	250K DISK/ 48K RAM
STORES MANAGEMENT PROCESSOR	8080	200 KOPS	32K CORE

TABLE III - SENSOR-ORIENTED ROM COMPUTERS

SUBSYSTEM	CPU	MEMORY
AIR DATA COMPUTER	2901	5K ROM
COMM SYSTEM CONTROLLER	8080	16K ROM
FLIGHT CONTROL COMPUTER (4)	MCP-701A	44K ROM
FORWARD-LOOKING INFRARED	9900	32K ROM
LASER SPOT TRACKER	2901	12K ROM
MAINTENANCE MONITOR PANEL	8080	1K ROM
MAINTENANCE SIGNAL DATA RECORDER SET	8080	14K ROM
MULTIPURPOSE DISPLAY (2)	2901	5K ROM

Each sensor computer performs only those computations necessary to perform its well-defined task. This includes all computations required to translate some measured physical parameter, such as air pressure, into useful information for the pilot, such as altitude, airspeed, and Mach number. Once the information is computed, it is sent to the Mission Computer over the Avionics Multiplex (MUX) bus. There it is used with information from other sensors to perform the mission-oriented computations as well as for display to the pilot. Figure (3) shows the major sensor computers and their allocated software computational tasks.

2.2 Mission-Oriented Processing

2.2.1 Mission Computer Hardware

The Mission Computer Subsystem consists of two identical computers built by Control Data Corporation (CDC). They are the new U.S. Navy Standard Airborne Computers designated the AN/AYK-14. Although the hardware of the two computers is identical, their computer programs are different and are dedicated to specific processing tasks. The AN/AYK-14 is a high-speed, general purpose digital computer specifically designed to meet the real-time requirements of airborne weapon systems. The computer uses four AMD 2901 four-bit slice Large Scale Integrated (LSI) circuits to implement the 16-bit Central Processing Unit (CPU). The CPU is micro-programmed by means of ROM firmware to emulate the instruction set of the U.S. Navy Standard Shipboard Computer designated the AN/UYK-20. By emulating the AN/UYK-20 instruction set, the AN/AYK-14 can use the same CMS-2M Higher Order Language (HOL) support software originally designed for the AN/UYK-20.

The AN/AYK-14 consists of ten plug-in modules and a single plug-in modular power supply contained in one Weapon Replaceable Assembly (WRA) weighing about 42 pounds and occupying 0.625 cubic feet. Each computer contains 65,536 (16-bit) words of 7/13 mil (inside/outside diameter) core memory for a total of more than a million individual cores per computer. The memory in each Mission Computer can be doubled from 64K to 128K within the present equipment envelope simply by replacing the two present 32K memory modules with two recently-developed 64K modules. Figure (4) shows the computer and some of its plug-in modules.

The salient features of the computer are presented below:

o Type and organization	General-purpose, stored program, parallel, binary, fixed-point, integer, two's complement
o Storage	65,536 words, 16 bits/word plus 2 bits/word parity, nonvolatile, random access, 3D magnetic core, 0.9 microsecond cycle time, 16 bit addressable
o Instruction execution rate	450,000 operations/sec (depending on instruction mix)
o CPU	Four AMD 2901 4-bit slice LSIs
o Instruction Set	ROM firmware emulation of AN/UYK-20 instruction set
o Serial Input/Output	Three independent, dual-bus MIL-STD-1553A multiplex channels (serial, 16 data bits plus 1 bit parity, transformer-coupled, 1 MHz, 50,000 words/sec/channel)
o Discrete Input/Output	32 input discretes 32 bi-directional input or output discretes
o Interrupts	Eight external, 22 internal
o Clocks	Two programmable clocks

2.2.2 Mission Computer Software

Each of the two Mission Computers is dedicated to specific processing tasks by means of its stored program. One computer is assigned the Navigation (NAV) and Support processing tasks and associated display management. The other computer is assigned the Air-to-Air and Air-to-Ground Weapon Delivery processing tasks and associated display management. The stored program in each computer is partitioned into functional software modules. Each software module is assigned to an engineer/programmer development team that follows the software module from initial concept until delivery of the computer program in the weapon system. Each computer has a small backup software module for selected functions of the other computer. These backup modules are executed only in the event the primary computer for these functions should fail. The functional software modules in each computer are shown in Figure (5).

2.2.2.1 Executive Module

The executive program module imposes order and structure on the entire F/A-18A operational flight program. All functional program modules are processed under executive control, which sequences them in an appropriate flow and calls them at a rate consistent with their requirements.

Six major tasks are performed by the executive module. First, it initializes the MC after start-up or after restart from a power interruption. Second, it schedules the order and rate of execution of each functional module. Third, it schedules the order and rate of input/output operations for each module. Fourth, it controls the servicing of all interrupts, external and internal. Fifth, it manages inter-computer communication between the Navigation MC and the Weapon Delivery MC. Sixth, it uses the scheduling and input/output management functions to ensure proper sequencing of applications processing.

2.2.2.2 Air-to-Air Module

The air-to-air module performs the following functions:

- 1) initializes the radar air-to-air search pattern based on the weapon selected
- 2) computes aiming reticle for director or disturbed gun mode
- 3) computes aiming reticle for director or manual rocket mode
- 4) computes maximum and minimum launch ranges and steering cues
- 5) computes other aircraft and target parameters for display.

2.2.2.3 Air-to-Ground Module

The air-to-ground module performs the following functions:

- 1) designates ground targets using radar, forward looking infrared (FLIR), laser spot tracker (LST), or visual means
- 2) automatically positions sensors
- 3) calculates ballistic release times
- 4) calculates steering cues for weapon release and reattack
- 5) calculates launch envelope data for air-to-ground missiles and gun
- 6) issues release pulses for correct weapon delivery and weapon intervals
- 7) manages strike camera (SCAM) for damage assessment.

2.2.2.4 Navigation Module

The navigation module performs the following functions:

- 1) selects/calculates the best available navigation data
- 2) calculates steering to prestored waypoints
- 3) performs velocity and position updates
- 4) performs target marking
- 5) calculates range, bearing, heading, and steering error to selected waypoint and TACAN station.

2.2.2.5 Data Link Module

The data link module decodes and processes messages received from a shipboard, airborne, or ground-based terminal. The messages contain information used in the following functions:

- 1) waypoint insertion
- 2) display of data for vectoring to airborne targets and rendezvous points
- 3) display of precision course direction data for air-to-ground weapon delivery
- 4) display of automatic carrier landing data
- 5) processing of couple requests to the flight control computers
- 6) processing of test messages
- 7) processing of radar target data and aircraft data to be transmitted in the data link reply messages.

2.2.2.6 Tactical Controls and Displays Module

The tactical controls and displays module performs the following functions:

- 1) manages the radar control panel/display graphics program. Symbology controlled by this function includes targets, target status, radar status, air-to-air weapon delivery cues, air-to-ground and navigation cues, armament status, aircraft flight status, data link targets and cues, pushbutton legends/status, and hands-on-throttle-and-stick cues. This function also controls radar and display moding so the above calligraphic symbology can be superimposed on radar video presentations.
- 2) manages the FLIR control panel/display graphics program. Symbology controlled by this function includes FLIR status, aircraft flight status, and pushbutton legends. This calligraphic symbology is superimposed on FLIR video presentations.
- 3) manages the LST/strike camera control panel/display graphics program. Symbology controlled by this function includes LST/strike camera status and pushbutton legends.
- 4) manages the air-to-ground guided weapons control panels/display graphics program. Symbology for the high-speed antiradiation missile (HARM), Maverick, and Walleye weapons controlled by this function includes targets, weapon status, aircraft flight status cues, weapon delivery cues, and pushbutton legends. This calligraphic symbology is superimposed on weapon video presentations.
- 5) manages the stores management control panel/display graphics program. Symbology controlled by this function includes stores status for each station, air-to-ground weapon delivery programming, and pushbutton legends.

2.2.2.7 Support Controls and Displays Module

The support controls and displays module performs the following functions:

- 1) manages the cautions/advisories display graphics program. Symbology controlled by this function includes cautions and advisories for engines, hydraulics, electrical, environmental control system, flight control set, and avionics systems.
- 2) manages the built-in test (BIT) display graphics program. Symbology controlled by this function includes avionic subsystem status, memory inspect data, maintenance panels, and pushbutton legends.
- 3) manages the test pattern display graphics program. Symbology controlled by this function includes test pattern, pushbutton test cues, and boresight cues.
- 4) manages the engine display graphics program. Symbology controlled by this function includes left and right engine status and pushbutton legends.
- 5) manages the checklist display graphics program. Symbology controlled by this function includes checklist cues, aircraft data, and pushbutton legends.

2.2.2.8 Navigation Controls and Displays Module

The navigation controls and displays module performs the following functions:

- 1) manages the horizontal situation display control panel/display graphics program. Symbology controlled by this function includes compass and associated steering cues, aircraft flight status, TACAN/waypoint data, alignment data, navigation data, update cues, and pushbutton legends.
- 2) manages the attitude director indicator display graphics program. Symbology controlled by this function includes aircraft flight status cues, such as attitude and turn rate.
- 3) manages the data link display graphics program. Symbology controlled by this function includes command data, data link cues, and data change cues.
- 4) manages the up-front control panel to provide data entry/readout and mode selection capability for autopilot, navigation data, and weapon delivery data.
- 5) computes the position of the film strip for moving map and navigation functions related to the horizontal situation display.

2.2.2.9 Head-Up Display Module

The head-up display (HUD) module manages the HUD graphics program. Symbology controlled by the HUD module includes aircraft flight data, data link cues, navigation cues, radar status, armament status, air-to-air weapon delivery cues, and air-to-ground weapon delivery cues.

2.2.2.10 Inflight Engine Condition Monitor Module

The inflight engine condition monitor module monitors various engine and associated aircraft parameters to provide engine health information to the pilot and maintenance personnel. Cautions, advisories, and real time engine parameters are displayed in the cockpit. Life usage indices and other engine maintenance information are transmitted to the Maintenance Signal Data Recorder (MSDR).

2.2.2.11 Inflight Monitoring and Recording Module

The inflight monitoring and recording module monitors and processes various aircraft sensor outputs for control and display of most of the pilot cautions and advisories and transmits avionic and non-avionic equipment failures to the MSDR. Control is provided for the data recorder and provision is made for the recording of tactical data in the air-to-air and air-to-ground aircraft modes. Also, aircraft fatigue levels are monitored and recorded during flight.

2.2.2.12 Avionics Built-In Test Module

The avionics built-in test module provides the control by which an operator can run individual tests on each of the interfacing subsystems. It also evaluates data received by the MC from each of the interfacing subsystems as to their operational status. This data is correlated by subsystem and current status and is displayed in the cockpit. In addition, the data is converted into predefined codes each representative of a specific failure of an individual subsystem for transmission to the MSDR.

2.2.2.13 Mission Computer Self-Test Module

The mission computer self-test module performs the following functions:

- 1) immediately after computer turn-on, tests those functions which, when tested, interfere with normal computer operation;
- 2) periodically tests those functions of the MC CPU and memory which, when tested, do not interfere with normal computer operation as well as performing an end-to-end check of the capability of the MC to communicate with each peripheral;
- 3) maintains error information for later maintenance action; and
- 4) latches WRA fault indicator and sets WRA status signal as required.

2.2.2.14 Mission Computer Backup Modules

A backup module is resident in each computer. Each backup module performs essential software functions of the other mission computer when a failure occurs in that computer.

2.2.2.15 Mathematical Subroutines Module

The mathematical subroutines module supports other program modules by providing common mathematical routines such as trigonometric, logarithmic, and matrix operations.

2.3 Avionics Multiplex System

Digital data between the Mission Computers and the peripheral avionics components is transferred on the MC-controlled Avionics Multiplex System. The system consists of three multiplex channels, as shown in Figure (6). Each channel consists of two redundant 1 MHz MIL-STD-1553A buses, with only one bus of each channel active at any given time.

2.3.1 Physical Characteristics

Each bus is operated in a half-duplex fashion (two-way transmission, but not simultaneously) using self-clocking Manchester encoding and word-serial, bit-serial, time-division format. All peripheral units on a single channel are connected to the transmission lines comprising that channel in parallel, party-line fashion, such that physical removal of a unit from the lines does not interrupt the continuity of the lines. All units on the same channel see all of the data on that pair of buses. However, on a given channel, data is transferred only between the MC and a single peripheral at a time. Each bus is independently routed through the aircraft to ensure reliable communication in the event of damage.

A multiplex terminal is incorporated as an integral part of each equipment interfaced with the MC. Each terminal performs the necessary functions to receive and validate data from the MC and transmit data to the MC. In addition, it provides the necessary conversion/reformatting of data to interface with the equipment component logic and accepts/generates the control signals that coordinate the transfers within the equipment. No peripheral is required to receive or transmit over more than one bus at a time. Data words transmitted by the MC or by the peripheral are always transmitted over the same bus that carried the command word.

All transmissions are formatted into standard messages. The MC initiates each message transmission by a command word that identifies the message, the number of data words, and the peripheral involved. Each word transferred contains a three-bit sync waveform, 16 information/control bits, and one parity bit.

2.3.2 Functional Characteristics

Only one of the buses of each redundant pair is active at any one time. The MC selects which of the data buses is to be used for data transmission and initiates each data exchange over the selected bus.

If the MC detects an input/output (I/O) error, i.e., no response from a peripheral, a parity error, or a data dropout, it will terminate processing of the I/O message. The MC then re-interrogates the peripheral by re-transmitting the same command word on the other bus. If another error occurs on the second bus, the MC internally flags an invalid response condition and proceeds to the next transmission scheduled for the peripherals on that channel.

Message transfer rates of 20, 10, 5, and 1 Hz are provided to match the execution rates of the modules which generate or use the data. On-demand transfers, such as weapon release, also are utilized. The Mission Computers have an independent I/O processor for the multiplex channels permitting full use of the computer CPU for processing tasks during I/O. However, transfers are controlled by the CPU to ensure that inputs, processing, and outputs occur sequentially for each rate. Control of the multiplex system is transferred between the two MCs based on priority of need.

3. MISSION COMPUTER SOFTWARE DEVELOPMENT

The F/A-18A Hornet software development process was based on testing the flight program before, during, and after the actual coding of the program. Figure (7) shows the five major phases of the software development.

- o Phase 1 consisted of creating FORTRAN models of selected equations, algorithms, and mode control. These models were tested in the Software Development Facility to provide the analytical validation of the equations and algorithms to be used in the OFP.
- o In Phase 2, a FORTRAN model of the baseline design was used at the McDonnell Douglas Cockpit Simulator Facility to evaluate the interface with the pilot and to test the mechanization proposed for the weapon system. This step provided vital confirmation of design adequacy at an early stage and allowed alternate approaches to be studied.
- o In Phase 3, the mathflows were coded in the CMS-2M language system and the program compiled in the Software Development Facility. The object programs were tested in the Software Test Facility. At this point, MC hardware/software inconsistencies were isolated and corrected, leading to preliminary confirmation of correct OFP software and MC equipment integration. The McDonnell Douglas Software Test Facility was used to monitor and control the integration.
- o As other avionics equipment arrived, Phase 4 tested the Mission Computer and associated software with actual interfacing equipment. This step provided integration of the MC and the MC OFP with the individual avionics equipment, followed by integration with groups of related equipment.
- o Phase 5 then reintroduced the man-in-the-loop to verify the total man/machine system. This phase used the McDonnell Douglas cockpit simulator with the OFP running in the Mission Computers along with the actual flight hardware for the controls and displays. The chief test pilot flew the first flight profile at the cockpit simulator prior to aircraft first flight. The MC OFP was then thoroughly evaluated during subsequent flight tests which are the final measure of its performance.

4. F/A-18A SOFTWARE DEVELOPMENT FACILITIES

The F/A-18A Hornet integrated software development process, discussed above, made use of three separate software facilities:

- o Software Development Facility
- o Software Test Facility
- o Cockpit Simulator Facility

4.1 Software Development Facility (SDF)

The Software Development Facility is a modest-size data processing facility. It uses an IBM System/370 commercial computer system and standard peripheral equipment, operating system, and language processors (see Figure 8)). This facility is used for all FORTRAN processing, database processing, and compilation/assembly of airborne MC programs.

Figure (9) is a block diagram of the Software Development Facility showing the IBM S/370 mainframe and associated peripherals. The facility includes the following equipment:

- o (1) IBM 370/138 Computer (512K memory)
- o (4) 100 megabyte disk drives
- o (2) magnetic tapes drives
- o (1) printer
- o (1) card reader
- o (5) CRT/keyboard terminals

The following software is currently active in the facility:

System Software

- o VM/CMS Operating System
- o IBM System/370 Assembler
- o FORTRAN H Compiler
- o FORTRAN H Library
- o SORT Utility
- o SCRIPT Word Processor
- o Display Editor
- o Plotter and Tablet Support
- o Graphics Attachment Support
- o Basic System Extension

Avionics Support Software

- o CMS-2M HOL Compiler/Assembler
- o MACRO-20/14 Assembler
- o CMS-2M SYSGEN
- o AN/AYK-14 Functional Simulator (SIM-14)
- o Avionics Environment, Sensor, and Display Models
- o Database Catalog Program
- o Operational Flight Program Tape Generator
- o PATHFIND Program
- o Display Compiler
- o Software Configuration Control Program

4.2 Software Test Facility (STF)

The Mission Computer Software Test Facility is a minicomputer-controlled, real-time simulation and test facility used to test the airborne Operational Flight Program (OFP) in the MC and to integrate the MC and its OFP with the other avionics with which they interface. The STF accomplishes this by simulating the inputs to the MC and sending them out over the Avionics MUX in response to the MC requests for data from various aircraft sensors. The MC processes these inputs as though it were flying in an aircraft and then issues output data to the simulated sensors and to the cockpit displays. In general, the input sensors are all modeled in software in the minicomputer whereas the CRTs used to display the MC outputs are the actual displays used in the cockpit. This provides a realistic input signal environment for the MC and a realistic display of MC outputs for test and evaluation by the engineers and programmers. Figure (10) is a block diagram of the STF.

4.2.1 STF Hardware

The hardware in the McDonnell Douglas STF is divided into three major benches plus the host computer system with its peripherals as identified below:

Host Computer Group (See Figure (11))

- (1) HARRIS/7 Minicomputer
- (2) Disk Storage Modules
- (1) Magnetic Tape Unit
- (1) Card Reader
- (1) Line Printer
- (4) CRTs (System Console and General User)
- (1) VERSATEC Printer/Plotter

General Purpose Interface Bench (See Figure (12))

- (1) High-Speed CRT and Keyboard
- (2) Simulation Control Panels
- (1) Analog/Discrete Interface Unit
- (1) Aircraft Stick/Throttle
- (1) Radar Interface Simulator

Avionics Integration Bench (See Figure (13))

- (1) Communications System Control (CSC)
- (1) Stores Management Snt (SMS)
- (1) Maintenance Signal Data Recorder (MSDR)
- (1) Head-Up Display (HUD)
- (1) Multipurpose Display Group (MDG) (three cockpit CRTs)
- (1) Up-Front Control

MC Integration Bench (See Figure (13))

- (2) Mission Computers
- (1) Multiplex and Discrete Interface
- (1) Control Keyboard
- (1) Magnetic Tape Drive
- (2) Lab CRT Displays
- (1) MUX Monitor and Peripheral Simulator
- (1) Interface for MC Support Channel

4.2.2 STF Software

The environment and avionics simulation software provides realistic real-time inputs for the Mission Computers by simulating the aircraft environment (e.g., Atmosphere, Equations of Motion) and the aircraft avionics subsystem (e.g., Radar, Air Data Computer, Inertial Navigation Set). There are four major divisions of these functions:

Scheduler

Aircraft Environment Modules

Atmosphere
Autotrim/Autopilot
Inertia, Forces, Moments
Equations of Motion
T . get
Aerodynamics

Aircraft Avionics Subsystems Modules

Radar
 Data Link (D/L)
 Inertial Navigation Set (INS)
 Air Data Computer (ADC)
 Laser Spot Tracker (LST)
 Forward Looking Infrared (FLIR)
 Flight Control Computer (FCC)
 Maintenance Signal Data Recorder (MSDR)
 (includes engine interface)
 Stores Management Set (SMS)
 Built-In Test (BIT)
 Communication System Controller (CSC)
 (includes interfaces for Instrument Landing System and TACAN)

Support Functions

Environment, Sensor, and Display Model Subroutines
 Input/Output Conversion Subroutines
 Aerodynamic Library Subroutines
 Multipurpose Display Group Subroutines

4.3 Cockpit Simulator Facility

The Cockpit Simulator Facility, Figure (14), is a laboratory complex oriented primarily to manned, real-time flight simulation. It includes a CDC Cyber 175 computer, four crew stations, terrain maps, horizon and target displays, and associated hardware. Each crew station includes complete flight controls and instruments and is located in a forty-foot fiberglass dome. Target and terrain imagery is projected on the dome and presented in the cockpit on software-driven displays or actual flight display equipment. Both visual and sensor (electro-optical, infrared, radar) imagery is supported. The facility is used for weapon system design, pilot training, tactics development, and effectiveness assessment.

5. SUMMARY

In summary, the F/A-18A Tactical Airborne Computational Subsystem is a distributed computer system. The mission-oriented computations are performed in two central Mission Computers and the sensor-oriented computations are performed in distributed processors in the sensor and display equipment. The memory in each Mission Computer can be doubled from 64K to 128K words within the present equipment envelope for a total combined capability of 256K words. This memory growth along with the flexibility of the MC multiplex input/output system and the distributed partitioning of the sensor and display computations makes the F/A-18A computational subsystem easily adaptable to changes and expansions in F/A-18A mission requirements and ready to share a long and successful future with the F/A-18A aircraft.

References

1. Griffith, V. V., Keifer, L. F., Paxhia, E. C., et al., "Aircraft Avionics Trade-Off Study (AATOS)," McDonnell Aircraft Co., St. Louis, Mo., ASD/XR 73-20 Final Report, Nov. 1973.
2. Fink, H. G. and Rosenkoetter, E., "Aircraft Avionics from the Aircraft Manufacturer's Point of View," McDonnell Aircraft Co., St. Louis, Mo., MCAIR 73-023, Sept. 1973.
3. McTigue, T. V., "F-15 Computational Subsystem," AIAA JOURNAL OF AIRCRAFT, Vol. 13, No. 12, Dec. 1976, Pp. 945-947.

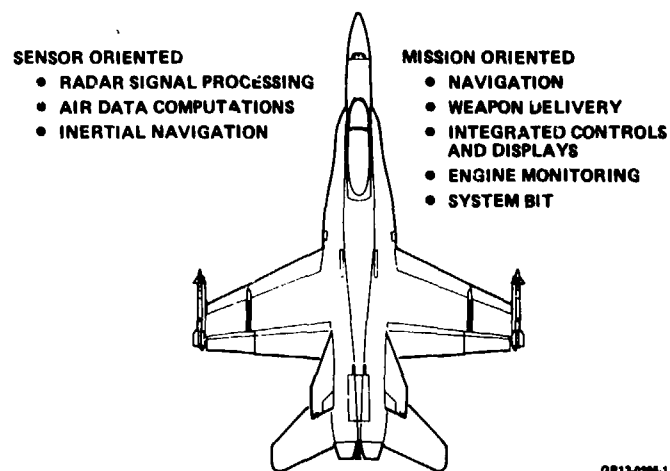


FIGURE 1
F/A-18A COMPUTATION REQUIREMENTS

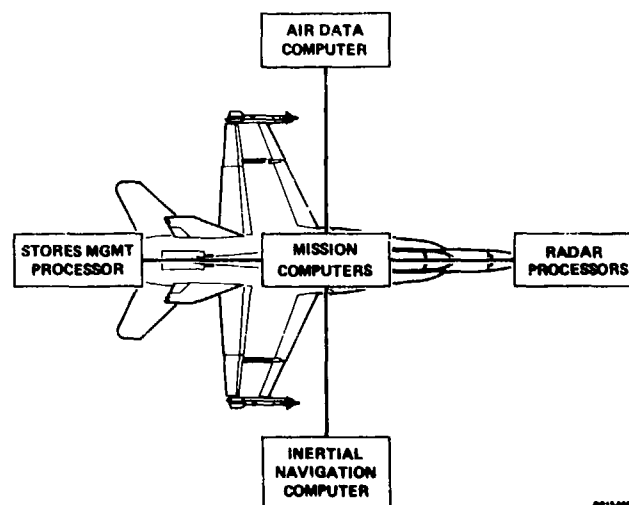


FIGURE 2
F/A-18A MISSION and SENSOR COMPUTATIONS

AIR DATA COMPUTER	INS COMPUTER	STORES MANAGEMENT PROCESSOR	RADAR SIGNAL PROCESSOR	RADAR PROCESSOR
<ul style="list-style-type: none"> • PRESSURES • AOA • SIDESLIP • ALTITUDE • AIRSPEED • MACH • TEMP • AIR DENSITY 	<ul style="list-style-type: none"> • ALIGN/GB • ACCELERATIONS • VELOCITIES • PRESENT POSITION • ATTITUDE 	<ul style="list-style-type: none"> • SPARROW INTERFACE • SIDEWINDER INTERFACE • GUN INTERFACE • BOMBS INTERFACE • HARM INTERFACE • WALLEYE INTERFACE • MAVERICK INTERFACE • RACK/VIDEO CONTROL • JETTISON • WEAPON INVENTORY 	<ul style="list-style-type: none"> • RCVR GAINS • SIGNAL THRESHOLDS • RANGE GATING • PULSE COMPRESSION • AMPLITUDE WEIGHTING • RANGE RESOLUTION • TARGET DETECTION • TRACK S/N 	<ul style="list-style-type: none"> • TARGET POSITION • TARGET VELOCITY • TARGET ACCELERATION • TARGET RANGE • VELOCITY ERRORS • DISPLAY DATA

GP03 0368-04

FIGURE 3
SENSOR-ORIENTED SOFTWARE FUNCTIONS

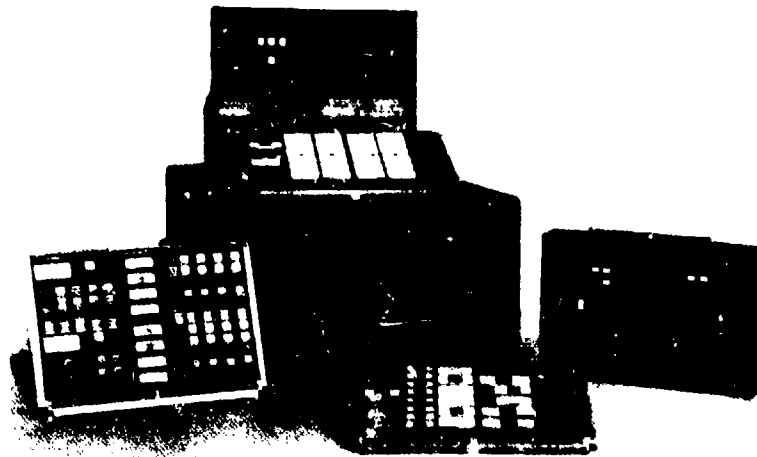


FIGURE 4
MISSION COMPUTER AND PLUG-IN MODULES

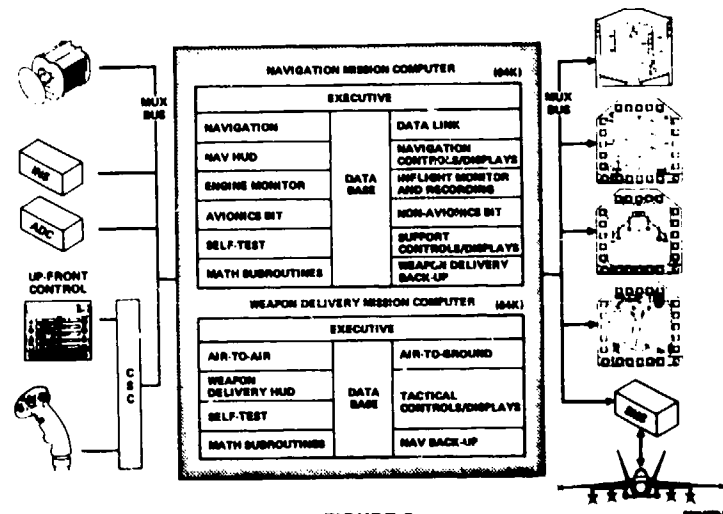


FIGURE 5
MISSION COMPUTER SUBSYSTEM

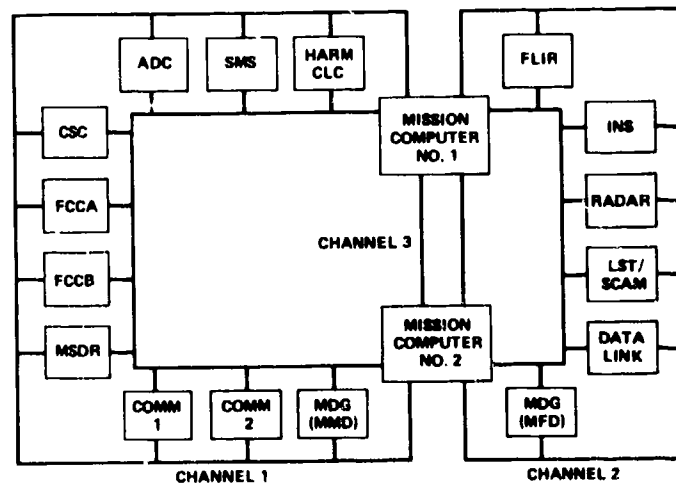


FIGURE 6
F/A-18A AVIONICS MULTIPLEX SYSTEM

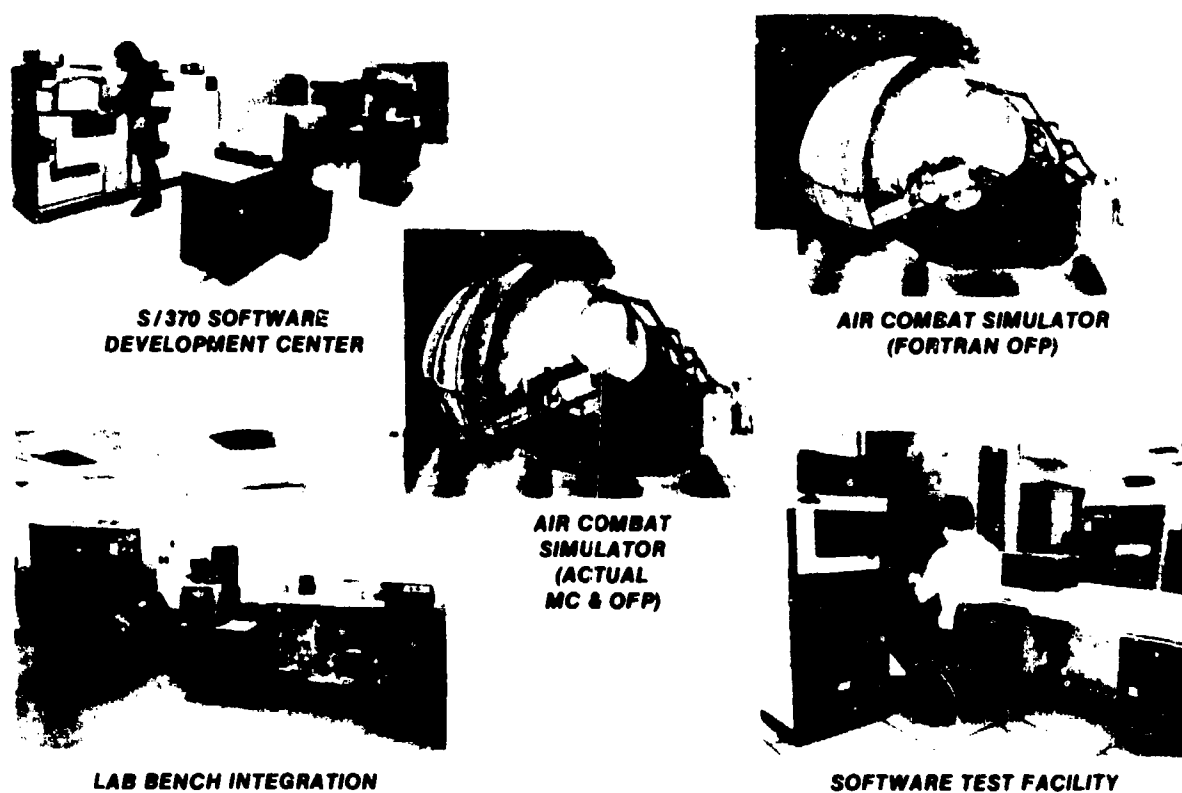
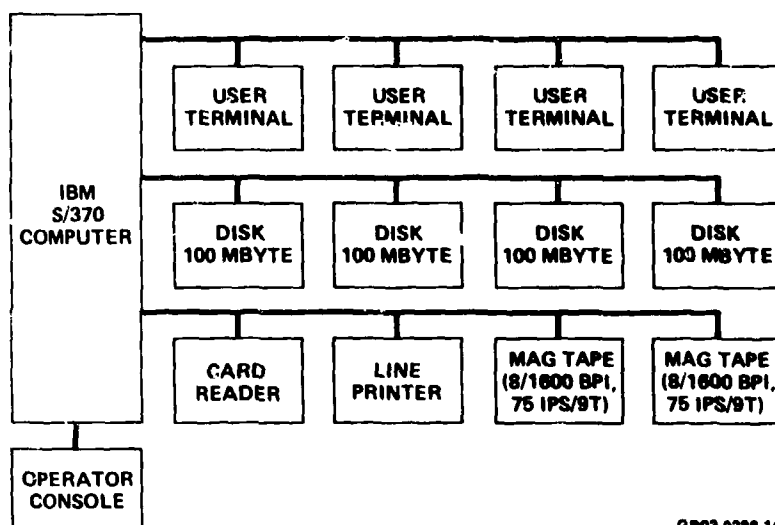


FIGURE 7
F/A-18A MISSION COMPUTER SOFTWARE DEVELOPMENT PROCESS

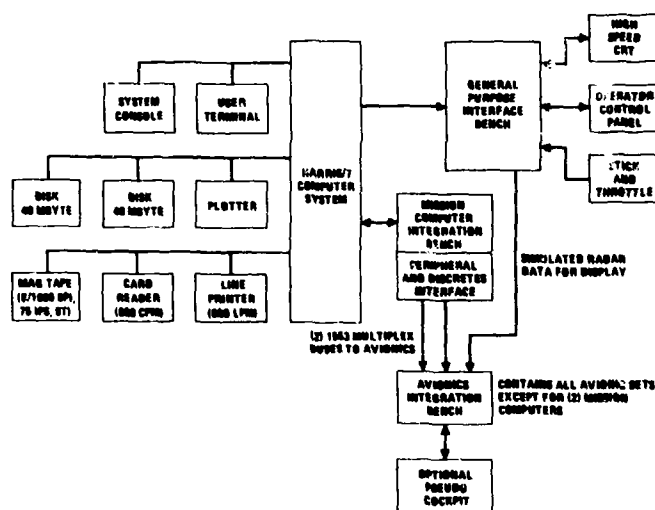


FIGURE 8
F/A-18A SOFTWARE DEVELOPMENT FACILITY



GP03-0384-148

FIGURE 9
F/A-18A SOFTWARE DEVELOPMENT FACILITY
BLOCK DIAGRAM



GP03-0384-149

FIGURE 10
F/A-18A SOFTWARE TEST FACILITY BLOCK DIAGRAM



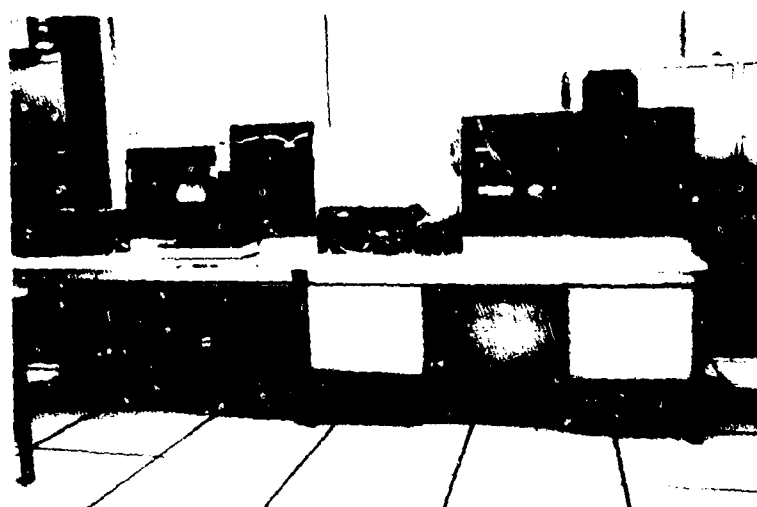
GP03 0384 4

FIGURE 11
F/A-18A SOFTWARE TEST FACILITY HOST COMPUTER GROUP



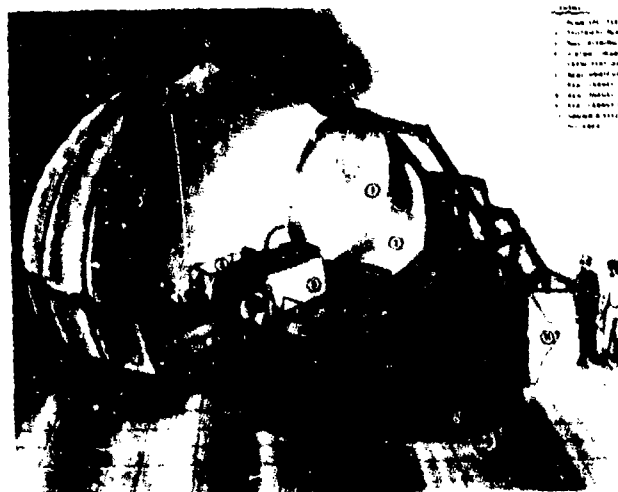
GP13-0000-150

FIGURE 12
F/A-18A SOFTWARE TEST FACILITY
GENERAL PURPOSE INTERFACE BENCH



GP13-0000-151

FIGURE 13
F/A-18A SOFTWARE TEST FACILITY
INTEGRATION BENCHES



GP13-0000-152

GP13-0000-153

FIGURE 14
F/A-18A COCKPIT SIMULATOR FACILITY

F/A-18 WEAPONS SYSTEM SUPPORT FACILITIES

Thomas F. O'Neill
 Naval Weapons Center
 F-13 Facility Branch (Code 3114)
 China Lake, CA 93555, U.S.A.

SUMMARY

The U.S. Navy is currently acceptance-testing the McDonnell Douglas F/A-18 aircraft. Since the F/A-18 is so much more complex than any aircraft currently deployed, more sophisticated support tools will be required. The main support tool will be a weapons system support facility. This facility will have all of the hardware and software necessary to test, modify, and validate all of the avionics hardware, software, and firmware. A distributed processing approach is used in the facility, which contains several minicomputers and super minicomputers.

1. INTRODUCTION

The McDonnell Douglas F/A-18 aircraft is an all-weather fighter/attack aircraft capable of hosting a wide range of ordnance. Figure 1.1 shows the F/A-18 with some of the ordnance it can deliver. Upon its acceptance into the Fleet, the Navy will assume responsibility for the modification, test, and certification of the avionics, software, and weapon systems in the aircraft. The Naval Weapons Center (NWC), China Lake, Calif., has the responsibility to provide system engineering, system integration, software development, configuration management, and test and evaluation support throughout the life cycle of the F/A-18 aircraft.

In late 1978, the Weapons System Support Activity was formed at NWC to provide system engineering support to the aircraft.

A specialized avionics support facility has been tasked by the Weapons System Support Activity to provide weapon system support during all phases of the weapon system life cycle.



FIGURE 1.1. The F/A-18 and some of its ordnance.

2. SUPPORTING THE F/A-18

With the addition of the F/A-18, NWC becomes the proving ground for an increasing majority of the fleet of fighter/attack aircraft within the Navy.

NWC has overcome a number of problems normally associated with verification and validation with the inception of a unique weapons system support facility. The long-term success of this support facility approach has been demonstrated in a variety of other programs, including those for the A-7, A-6, A-4, and AV-8B aircraft.

Validation through the use of test flights is impractical because of time and cost factors. By providing a work station in which the various subsystems of the aircraft can be modified and rigorously tested, NWC has succeeded in reducing cost and manpower requirements dramatically.

2.1 The Weapons System Support Facility

During the life cycle of the F/A-18, many changes will have to be made to the avionics software. These changes may be made in response to problems detected or new capabilities desired by the Fleet, or may occur with the addition of new equipment to the aircraft.

The Weapons System Support Facility (WSSF) will contain the avionics processors, commercially available computers, and the hardware and software necessary to test the operational flight programs. The facility then becomes the main tool for validation. Any change to the avionics equipment will be tested in the facility before its incorporation into the aircraft.

The facility will provide two broad categories of software—simulation and support. The simulation is a high order language program package that, given the same inputs, will produce an output identical to that of the avionics subsystems. The facility user can then work with a mixture of simulated and real avionics. The support software consists of programs that allow the engineer to generate, test, and validate new load modules for the avionics computers.

Since the avionics computers are not designed for software development, the facility's computers must provide the ability to modify the source code for the flight programs, compile or assemble them, and then form a load module that the avionics computers can utilize. The tools necessary to implement this process include cross compilers and cross assemblers to translate the source code into an object file, and some form of linker/loader to create a load module from the individual object code files.

2.2 The F/A-18 Aircraft

The F/A-18 is a sophisticated, high-performance aircraft that is, in itself, a distributed processing system. There are approximately 30 computers with a total of 700,000 words of program storage in the aircraft. These computers range in size from microprocessors with their programs in read-only memory to general-purpose computers with more than 256K words random-access memory and disk memory. Each subsystem in the aircraft (for example, the inertial navigation system, stores management set, and the radar) is a separate, self-contained computer that uses a dual redundant MIL-STD 1553 bus to communicate with the two AYK-14 mission computers.

The AYK-14s act as bus controllers while the other computers in the aircraft respond to the commands from the AYK-14 as remote terminals. That is, the AYK-14s act as "traffic directors"; all data on the 1553 either comes from or goes to the AYK-14s.

The F/A-18 cockpit is designed to give the pilot a visual display of all information concerning the operation of the aircraft. A control panel could not be provided for each subsystem because of the prohibitive number of subsystems present within the aircraft. Therefore, the solution was to place three cathode ray tubes in the cockpit, and drive these displays by two microprocessor generators. These displays are surrounded by 20 pushbutton switches, which are used to select the information to be displayed, change the modes of the avionics computers, and select the weapons to be dropped. Typical displays are shown in Figure 2.1.

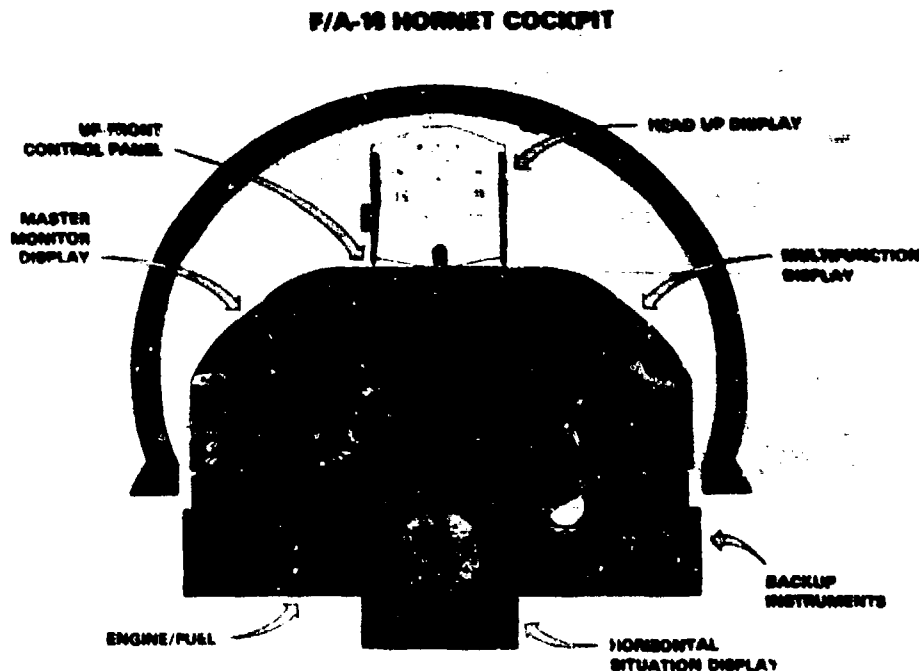


FIGURE 2.1. The F/A-18 cockpit.

2.3. Functional Requirements

The following are functional requirements to which the Weapons System Support Facility must respond:

(a) The facility must provide the ability to interchange simulated models of the avionics subsystems with the real avionics hardware. This must be accomplished in such a manner that the real avionics hardware cannot detect that the rest of the aircraft, as well as the world, is being simulated.

(b) The 1553 bus traffic has to be realistic. This requires that the hardware interface between the facility computers and the 1553 has to respond in the prescribed manner, and that the data that the simulation generates must be in the correct format.

(c) The software written for the simulation must be in some high order language (HOL). The facility is expected to support the F/A-18 program well into the 1990s. Over the life of the laboratory, if the software is not readable and easily modified, the cost would become astronomical. This requirement has a major impact on the design of the laboratory because the avionics software is mostly coded in assembly language. The amount of memory and time it would take for an HOL simulation of assembly language programs of this size becomes a major concern.

(d) In order to host the cross assemblers, cross compilers, and the linker/loaders necessary to develop software for the avionics computers, a large address space is essential in the facility's computers. These programs have been developed by the contractors who produce the avionics subsystems, and are currently hosted on IBM mainframes.

(e) Line printers, magnetic tape drives, large disk storage, and graphic devices all must be provided in the facility to store and process the data collected from the simulation and from flights.

3. A DISTRIBUTED PROCESSING APPROACH

Analysis of the above requirements revealed two basic approaches to the WSSF design.

The mainframe approach requires the purchase of a single computer to host the simulation and all associated tools.

The distributed minicomputer design involves the use of a number of minicomputers tied together in a distributed network scheme.

3.1. Advantages of the Distributed Approach

Information gathered from other facilities revealed that the distributed approach has several advantages over the single mainframe.

There are typically three types of users who need access to the simulation computers:

The simulation programmer requires computer time to code, test, debug, and integrate his software with the other models in the simulation.

The hardware engineer needs access to the computer to interface and test new equipment.

The simulation user uses the computers to run the programmer's simulation using the engineer's hardware.

In the past, limiting users to a single computer quickly resulted in scheduling problems. Multiple computers solve this problem by dedicating an individual computer to each particular area of need.

The distributed processing approach also has the advantage of easily accommodating the addition of more minicomputers to meet future needs. When dealing with a project the size and complexity of the F/A-18, it is impossible to accurately access future needs, particularly in the area of computer memory requirements. The use of minicomputers provides an unlimited expansion capability.

3.2. Selection of a Family of Computers

The Digital Equipment Corporation's (DEC) PDP-11 line of computers has been chosen as the architectural base of the facility because it offers a broad range of computers that can meet the general support and real-time requirements in the facility. In addition, DEC supplies a networking scheme which is extremely applicable to the facility's distributed processing approach.

3.3. Testing Tools

A necessary function of the facility is to provide a means of testing the avionics software. The primary tool to provide this capability will be a simulated environment that, using a combination of software models and real avionics subsystems, appears to the avionics computers as an F/A-18 aircraft in flight. The simulated models have to provide all of the necessary outputs in the correct format to stimulate the other models and any real avionics; if the real avionics needs sensory inputs, the simulation computers will have to provide these stimulations.

In addition, the facility provides a macro-level emulation of the avionics computers in the form of a software package that, using the load module of the avionics computers as input, emulates the actions of the avionics computers one macro-code instruction at a time. This package also has the ability to set breakpoints in the execution to allow the operator to examine the data within the program and trace the path of the program through the load module.

When the simulation is running, data will be passed from one model to another within the simulation computers and from one avionics computer to another on the 1553 bus. The facility computers will provide real-time monitoring of selected subsets of all of this data.

3.4. Evaluation Tools

Software packages are provided to allow the system engineers to evaluate the simulation and avionics software. Specifically:

(a) Any or all of the data that is passed among computers on the 1553 bus can be recorded on magnetic media for later data analysis.

(b) By recording the control inputs to the simulation and passing these back through the simulation at a later date, the simulation can be forced through the same maneuvers time after time. Using this method, differences between two versions of the avionics software can be detected.

(c) The 1553 data can be recorded while the aircraft is in flight and used at a later date to drive the simulator cockpit displays.

(d) The facility will provide a wide range of data-reduction software, from line printer listings to user-interactive, plotting packages. These data-reduction techniques allow rigorous scrutiny of data collected during flight for the purpose of isolating errors and specific problem areas.

3.5. General Support Capability

Because of the large address space required by certain software packages, such as cross assemblers and cross compilers, a PDP VAX 11/780 was purchased. The VAX is a 32-bit, virtual memory computer with disks, magnetic tape drives, and line printers needed to support the avionics software generation tools. A backend graphics processor has been added to facilitate data reduction. This relieves the VAX of the heavy processor load normally associated with graphics software. Figure 3.1 is a schematic layout of the VAXs used in the facility.

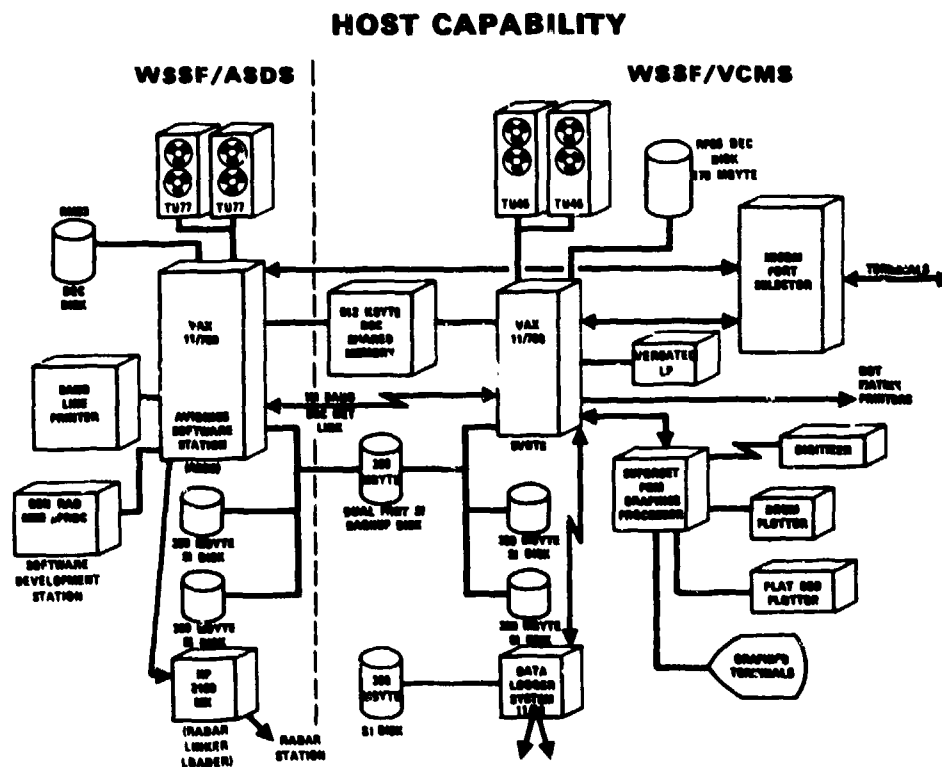


FIGURE 3.1. PDP VAX 11/780 schematics.

3.6. Real-Time Stations

In the area of general support, the only requirement is that the work be completed. There are, however, real-time aspects of the facility whereby the work must be completed within a certain time frame. These real-time requirements indicated that three types of work stations would be required--the integration, validation, and special function stations.

3.6.1. The Integration Work Station

Error correction or the addition of hardware requires that a change be made to the avionics programs. The initial testing of any modification is made in the integration station. If a change must be made to the avionics software or hardware, the engineer will devise a solution, be it a simple software fix or something as complicated as a new weapon system. The integration station will be used to test this modification until the performance meets with the engineer's approval.

3.6.2. The Validation Work Station

Once the software or hardware has passed the initial testing in the integration station, it is transferred to the validation station. Just as the name implies, validation involves rigorous testing for the purpose of ensuring that the modification made has corrected the known errors without introducing any additional ones.

A sixth computer drives the display units. Since the display group has a very powerful instruction set, one 11/60 is dedicated to processing the display data received on the 1553 bus from the AYK-14s.

Yet another 11/60 is dedicated to hardware development. The hardware engineer can develop his interfaces on this machine, and if the device ever crashes the computer, he can simply reboot without affecting other users.

4.3. Multiport Memory

Because the simulation tasks are not all located within the same computer, there has to be a means of communicating data from one model in one computer to another model in a different computer. Conventional communication schemes were not appealing since they were relatively slow (on the order of milliseconds per transfer). To solve this problem, a multiport memory has been designed and fabricated.

This memory unit is 8192 words of random-access memory which contains eight computer ports. Each port can interface with a different computer, thereby allowing up to eight computers to communicate with each other at memory access speeds (on the order of microseconds, a factor of 1000 better than other communication schemes). This memory is fast enough so that if all eight ports request data simultaneously, all requests will be granted within the normal memory access speed of the 11/60 memory system. There is a self-contained arbitrator in the common memory unit to resolve multiple, simultaneous data requests.

Error logging and diagnostic ports have been built in to facilitate debugging of both hardware and software errors. These ports allow errors to be collected for detection of error-causing conditions within the memory system.

4.4 The Displays

The simulation of the two display microprocessors is complicated by the addition of an out-the-window display behind the head-up display. This out-the-window view has been added to increase the realism of the work station. Figure 4.2 shows the three displays and the out-the-window view.

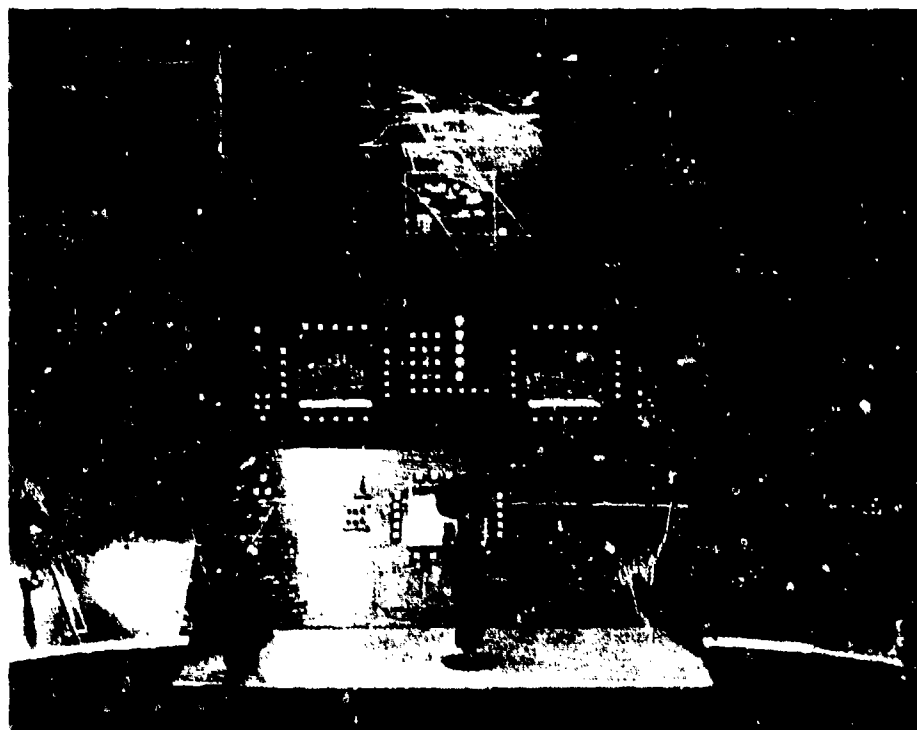


FIGURE 4.2. The integration cockpit.

The test flights that are currently planned for the F/A-18 when it arrives at NWC will cover several thousand square miles. To be able to display all of the areas the pilot will be looking at during these flights, software has been written to digitize the background data for the western half of the United States. Data is constantly being added to the background to increase the detail.

The display of the graphic data (the background and the simulation of the avionics displays) is divided between two graphical processors. The background is displayed by an Evans & Sutherland Picture System II. This is a highly capable graphics system that displays data in three dimensions, with the processor handling the time-consuming work involved in the generation of three-dimensional data.

The simulation of the avionics displays is performed on an ADAGE 4145, which is a two-dimensional device having the features of a user-programmable writable control store (WCS). The WCS can then be programmed so that the ADAGE performs as though it were two avionics display generators.

4.5. The Static Control Panel

Figure 4.3 shows another integral part of the cockpit work station, the static control panel. This "static panel" is a series of control switches and light-emitting diode displays which allow operator control over certain parameters affecting the aircraft. By dialing in the roll, pitch, and heading of the aircraft, the user can effectively fly the simulator from this panel. The simulation is constantly monitoring the panel to see if the operator has selected control of any of the more than 30 parameters available. If a particular variable is not selected for operator control, the simulation will calculate it; if on the other hand, the operator does want control, then the simulation receives the value for the parameter from the panel inputs. In this way, the aircraft can be frozen in space and any of the parameters can be varied in small, controlled steps.

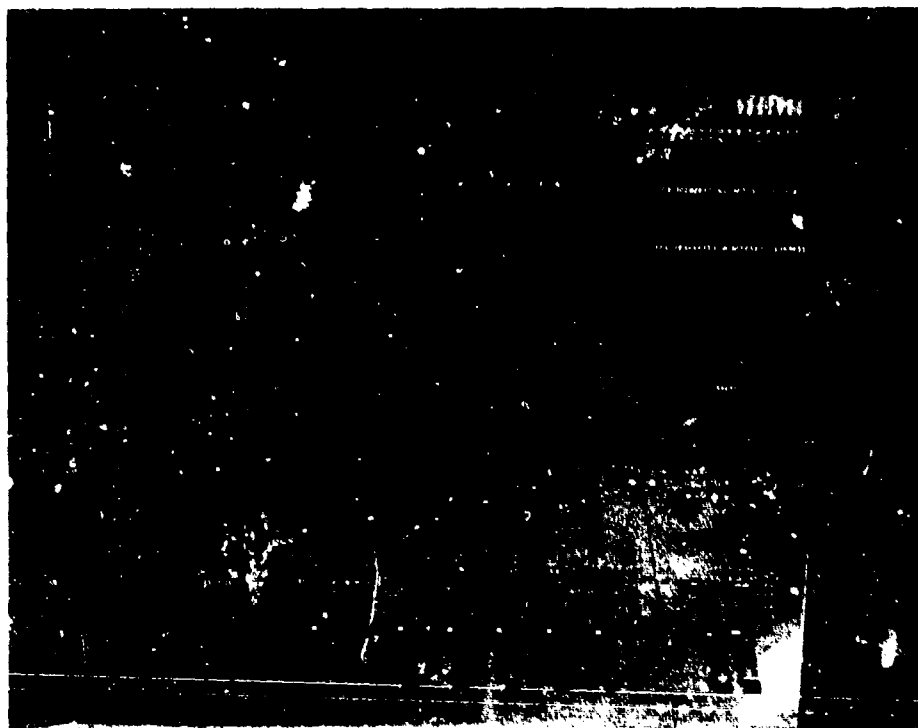


FIGURE 4.3 The static panel.

4.6. Data Logging

Hardware and software will have to be built and written to interface with the 1553 bus and collect all or any part of the data that is being passed from one computer to another. The data will be collected, time tagged, and then recorded directly onto magnetic media for later analysis.

5. CONCLUSION AND SUMMARY

The Naval Weapons Center has a long and successful history of supporting aircraft using the Weapons System Support Facility approach. Facilities to support the newer avionics systems are required to be more complex, cost effective, and support a project from initial verification through Fleet maintenance. By providing a WSSF for the F/A-18 aircraft, NWC will succeed in reducing long-term cost and manpower requirements dramatically.

DISCUSSIONS
SESSION VII

REFERENCE NO. OF PAPER: VII-32

DISCUSSOR'S NAME: Alan Stern, Boeing Co.

AUTHOR'S NAME: K. Moses

COMMENT: In the event of an in-flight reconfiguration, how is new software loaded and how is it assured to be correct?

AUTHOR'S REPLY: In-flight reprogramming of software (as distinct from reconfiguration of LRU's) is not contemplated.

REFERENCE NO. OF PAPER: VII-32

DISCUSSOR'S NAME: Dr. A. A. Callaway, RAE

AUTHOR'S NAME: K. Moses

COMMENT: One of the rumors we hear coming out of USA is the requirement by the USAF that the MIL STD 1750A Instruction Set Architecture be used for all future embedded computer applications. Is the SIFT concept compatible with the 1750A ISA?

AUTHOR'S REPLY: The SIFT concept is compatible with the 1750A ISA. The problem might be to find the processor that has the 1750A ISA and the required speed for flight control computers. We, in the US, hear the same rumor, and hope that it's not true.

REFERENCE NO. OF PAPER: VII-33

DISCUSSOR'S NAME: G. Hall, Sweden

AUTHOR'S NAME: Nelson

COMMENT: Have I understood it right that this can be used only on machine-language level?

AUTHOR'S REPLY: Yes, for those functions that involve instruction by instruction analysis. However, many of the functions are more global in nature. SOVAC could use information from a compiler symbol table and loader map to allow most functions to be used with high-level languages.

In fact, if the SOVAC software had full access to the data available from a compatible high-level compiler, it could perform all corresponding SOVAC functions for the high-level language user.

REFERENCE NO. OF PAPER: VII-33

DISCUSSOR'S NAME: Richard Schwartz, SRI, USA

AUTHOR'S NAME: H. Nelson

COMMENT: In what language is your application software written? As you move to higher level languages, do you expect SOVAC to still be useful? Would you attempt to support design aids taking advantage of the use of a higher level language?

AUTHOR'S REPLY: (1) The SOVAC software is written in PASCAL. (2) SOVAC has the ability to recognize and collect data on all observable activity in the tactical computer. Thus, it is primarily an issue of SOVAC software development to give it high-level language support capability. (3) In the future, we plan to develop SOVACs for machines using high-level languages. As we develop the requirements, we will investigate reasonable high-level aids. I expect it will be able to fully support the needs of the high-level language user.

REFERENCE NO. OF PAPER: VII-33

DISCUSSOR'S NAME: M. Mansell, British Aerospace

AUTHOR'S NAME: Harvey G. Nelson

COMMENT: On a number of occasions during flight development trials on Sea Harrier we experienced core store corruptions in the main computer. Have you experienced similar problems at the NWC and could you describe whether and how the SOVAC system could be used to investigate such problems?

AUTHOR'S REPLY: We have had similar types of problems at NWC. SOVAC was developed to be able to assist in the isolation of the source of these faults. To identify the memory locations changed, use the verify function to compare all protected locations against the corresponding values in a reference

file. Once the locations have been identified SOVAC can be used to watch activity associated with those locations. Any of the SOVAC functions can be used to collect the data desired to analyze the cause of the problem.

REFERENCE NO. OF PAPER: VII-33

DISCUSSOR'S NAME: W. R. Richards, Smith Industries

AUTHOR'S NAME: H. Nelson

COMMENT: I am familiar with the facilities provided by Universal Microprocessor Development Systems--in particular the Tektronix 8002 system. The facilities provided by "SOVAC" seem almost identical. Would the speaker please comment?

AUTHOR'S REPLY: Microprocessor development systems address the same general programmer needs. However, they are intended to be used with specific microprocessors with complete access to all needed signal and data lines. Also, their capability is somewhat low in bandwidth.

SOVAC is designed to work with high capacity, high speed "mini" computers with less than full access to the desired signal and data lines. SOVAC has a very high bandwidth and high capacity data collection capability.

Finally, SOVAC has the full data collection, storage, and computational capabilities of the PDP 11/34 computer to support it. Most enhancements to SOVAC involve only changes to the SOVAC software in the PDP 11/34.

REFERENCE NO. OF PAPER: VII-34

DISCUSSOR'S NAME: Jim McCuen, Hughes, USA

AUTHOR'S NAME: G. Wilcock

COMMENT: Has the UK as yet developed a MIL-STD for Solid State Power Controllers (SSPC)? Is there any SSPC in production?

AUTHOR'S REPLY: The UK has not yet produced a document similar in scope to a MIL standard for Solid State Power Controllers, although standardization is being pursued in a number of different ways. Specifications EL2141 and EL2143 have been published (source Elec 2/4, MOD(PE)). These do not have the authority of a MIL standard but are intended to promote information interchange between prospective users and suppliers and subsequently to serve as a basis for particular equipment specifications. The UK MOD is participating in the activity of the International Standards Organization which is drafting a standard for Remote Power Controllers (ISO/TC20/SC1). British Defence Standard 00-18 (Part 4)/Issue 1 relates to discreet (on/off) signaling but includes the operation of controllers for load ratings up to 0.2A. A more comprehensive coverage of loads and ratings is being considered for future issues.

There are no Defence Standard SSPCs in production, although there are a number of different devices in an advanced state of development. UK MOD has funded development of both ac and dc solid state and hybrid units at Plessey, Titchfield. Development of a monolithic IC to perform the control functions of an SSPC with increased reliability and reduced volume is also being funded (Swindon Silicon Systems, Swindon). Initial devices are being evaluated.

REFERENCE NO. OF PAPER: VII-35

DISCUSSOR'S NAME: Schoelch, IABG

AUTHOR'S NAME: S. Croce

COMMENT: Can you give some figures on the size of the software programs, especially of the main computer? Do you have any experience in software maintenance of this system? Which people and how many people are involved in this business?

AUTHOR'S REPLY: (1) Les programmes du calculateur principal occupent entre 40 et 50K mots pour une mémoire installée de 64K mots de 16 bits (plus 2 bits de parité). (2) Le système du mirage 2000 est actuellement en cours de développement. On procède donc à des modifications plutôt qu'à de la maintenance. Celle-ci sera cependant effectuée par les fabricants des matériels, qui sont aussi les "fabricants" du logiciel incorporé. Cette maintenance sera toutefois facilitée par l'utilisation d'un langage de haut niveau (Itr) et la mise en pratique d'une méthodologie rigoureuse qui oblige les programmeurs à réaliser la documentation en même temps que le codage.

(1) The programs of the main processor occupy from 40 to 50K words for an installed memory of 64K words of 16 bits (plus 2 parity bits). (2) The system in the Mirage 2000 is presently in the process of development the next (step) is to proceed to (a phase of) modifications rather than to maintenance. The latter (maintenance), moreover, will be accomplished by the equipment manufacturers, who are also the "manufacturers" of the embedded software. The maintenance will be ever facilitated by the use of a high order language (HOL) and the application of a rigorous methodology which forces the programmers to do the documenting at the same time as the coding.

REFERENCE NO. OF PAPER: VII-35

DISCUSSOR'S NAME: M. Mansell, British Aerospace, Kingston Division

AUTHOR'S NAME: B. Vandecasteele

COMMENT: On your dynamic development rig do you inject "dynamic" computer generated synthetic signals representing a target and if you do, at what point do you inject these into the radar system?

AUTHOR'S REPLY: La stimulation peut s'effectuer de deux manieres différentes:

- le radar poursuit effectivement une cible réelle; ses informations sont alors traitées par les équipements du banc et peuvent être envoyées à l'autodirecteur du missile.
- le radar ne poursuit pas de cibles; la mission est entièrement simulée. Les échos simulés sont alors injectés au niveau de la ligne numérique interne du radar.

Stimulation can occur two different ways:

- The radar effectively tracks a real target; its data are then processed by the test-bench equipment and can be sent to the missile's automatic controller.
- The radar does not track targets; the mission is entirely simulated. Simulated echos are then injected at the level of the digital data line internal to the radar.

REFERENCE NO. OF PAPER: VII-38

DISCUSSOR'S NAME: M. Mansell, British Aerospace, Kingston Division

AUTHOR'S NAME: T. F. O'Neill

COMMENT: If you wish to look at particular parameters within a mission computer computation do you make changes to the OFP to output data for recording and how do you cope with flight clearance of the OFP with this mode in and then removed when the problem has been solved?

AUTHOR'S REPLY: Two choices: (1) put the patch in, validate it and leave it in. This is useful only if the problem is a long term one. (2) put patch in, solve the current problem, take patch out, then validate.

REFERENCE NO. OF PAPER: VII-38

DISCUSSOR'S NAME: G. Scotti, SELENIA, Italy

AUTHOR'S NAME: T. F. O'Neill

COMMENT: How many people are currently joining the WSSF team? And how much man-year effort have you spent on the F18 program?

AUTHOR'S REPLY: There are 50-55 people working full time for the WSSF. Totally, there are probably twice that at NWC.

APPENDIX

LIST OF ATTENDEES

ACTON, A.A. Mr	Marconi Avionics (Training Dept.) Ltd, Airport Works, Rochester, Kent ME1 2XX, UK
ATKINS, R.J. Mr	Smith Industries, Aerospace & Defence Systems Co., Winchester Road, Basingstoke, Hants, UK
BAHRE, R. Mr	Fraunhofer-Institut für Informations, u. Datenverarbeitung, Sebastian-Kneipp Str. 12-14, D-7500 Karlsruhe, FRG
BALL Wm.F. Mr	Head, Avionics Facilities Div., Naval Weapons Center (Code 311), Dept. of the Navy, China Lake, CA 93555, USA
BARBER, B. Mr	ADV Team, British Aerospace P.L.C., Aircraft Group-Warton Division, Warton Aerodrome, Preston, Lancs, UK
BARTH-NILSEN, K.W. Mr	A/S Kongsberg Vaapenfabrik, Boks 25, N-3601 Kongsberg, Norway
BENNIS, H.G.M. Mr	Physics Laboratory TNO, Oude Waalsdorperweg 63, The Hague, The Netherlands.
BRAATHE, R. Mr	A/S Kongsberg Vaapenfabrik, Boks 25, N-3601 Kongsberg, Norway
BRAMMER, K. Dr	ESG Elektronik-System-Gesellschaft, Postfach 800569, D-8000 München 80, FRG
BRAULT, Y. Mr	Sous-Directeur, Thomson-CSF, 178 Bd Gabriel Péri, 92240 Malakoff, France
BROSS, P.A. Mr	Postfach 80 05 69, Electronic-System-GmbH, Vogelweideplatz 9, D-8000 München 80, FRG
CALLAWAY, A.A. Dr	Flight Systems Dept. Y20 Bldg., Royal Aircraft Establishment, Farnborough, Hants GU14 6TD, UK
CLEMENT, Mr	Ferranti Ltd, Ferry Rd., Silverknowles, Edinburgh EH4 4AD, UK
CROVELLA, C. Mr	Caselle Plant Manager, AERITALIA-Gruppo Equipaggiamenti, Esercizio di Caselle, 10072 Caselle Torinese, Italy
DANIEL, Mr	Thomson CSF, 52 rue Guynemer, 92130 Issy les Moulineaux, France
DELEGUE, Mr	Thomson CSF, 52 rue Guynemer, 92130 Issy les Moulineaux, France
DE WINTER, J. Capt.	Belgian Airstaff - VDT/B, Rue d'Evere 1, 1140 Brussels, Belgium
DIAMOND, F. Dr	Chief Scientist, Rome Air Development Ctr./CA, Griffiss AFB, NY 13441, USA
DOVE, B.L. Mr	Head, Avionics Systems Branch, Electronics Directorate, Mail S. 477, NASA Langley Research Center, Hampton, VA 23665, USA
DUKE, P. Mr	British Aerospace Aircraft Group, Kingston/Brough Division, Brough, North Humberside HU15 1EQ, UK
DUNCAN, I. Mr	Ferranti Ltd, Ferry Road, Silver Knowles, Edinburgh, Scotland, EH5 2XS, UK
EIKELAND, G. Major	Air Material Command, P.O. Box 10, 2007 Kjeller, Norway
EVANS, B. Mr	Marconi Avionics Limited, Elstree Way, Borehamwood, Herts, UK
FAEGRI, A. Mr	A/S Kongsberg Vaapenfabrik, Boks 25, N-3601 Kongsberg, Norway
FANTOZZI, C. Ing.	Industrie Face Standard, Via Della Magione, 00040 Pomezia, Roma, Italy
FERRERI, J.F. Mr	Avions Marcel-Dassault, 78 Quai Carnot, 92214 St. Cloud, France
FORGUES, M. Mr	CIMSA, 10-12 Ave de l'Europe, 78140 Velizy, France
GANGL, E.C. Mr	ASD/ENAI, Wright-Patterson AFB, Dayton, Ohio 45433, USA
GERHARDT, L. Prof.	Systems Engineering, Rensselaer Polytechnic Institute, Troy, N.Y. 12181, USA
GHICOPOULOS, B. Mr	Hellenic Air Force Technology, Research Centre (KETA), Delta Falirou, P Faliron, Athens, Greece

GIORDANI, E. Dr	Systems Engineering Mgr., c/o S.I.A., Via Canova, 25, I 10126 Torino, Italy
GOULET, Mr	MATRA, BP No. 1 - Ave. Lous Breguet, 78146 Velizy, Villacoublay Cedex, France
GREFFET, R. Mr	SFIM, 13 Ave. Marcel Ramolfo-Garnier, 91301-Massy, France
GRICE, J.A. Mr	Attn of TRC/Personnel Dept., Easams Ltd, Lyon Way, Frimley Road, Camberley, Surrey GU17 0P4
HALL, L.G. Mr	Research Institute of National Defence, Dept. 2, Fack, S-10450 Stockholm, Sweden
HARDENBOL, A.G. Ir.	Scientific Advisor to Cincent, HQ AFCENT, Post Box 270, 6440AG, Brunssum, The Netherlands
HARTKE, Dipl. Ing.	Institut für Luft-und Raumfahrt, Marchstr. 14, Skr. F3, D-1000 Berlin 10, FRG
HAUGLAND, T. Mr	N.D.R.E., P.O. Box 25, Kjeller, Norway
HEGER, D.	Fraunhofer-Institut für Information, u. Datenverarbeitung, Sebastian-Kneipp-Strasse 12/14, D-7500 Karlsruhe, FRG
HELPS, Mr	Smiths Industries, Aerospace & Defence Systems Co., Cheltenham Div., Bishop's Cleeve, Cheltenham, Glos. GL52 4SF, UK
HOENINK, G. Mr	Centrum Automatisering Wapen en Commandosystemen, Koninhlyke Marine, Marine Postkastoor, 1780 CA Den Helder, The Netherlands
HOIVIK, L. Dr	NDRE, P.O. Box No.25, N-2007 Kjeller, Norway
HUNT, G.H. Dr	Royal Aircraft Establishment, Farnborough, Hants GU14 6TD, UK
HVINDEN, O. Mr	N.D.R.E., P.O. Box 25, N-2007 Kjeller, Norway
von ISSENDORFF, H. Dr	Forschungsinstitut Funk & Mathematik, Konigstr. 2, D-5307 Wachtberg-Werthhoven, FRG
JACOBSEN, M. Mr	AEG-Telefunken N14/V3, D-7900 Ulm, Postfach 1730, FRG
JANIK, K. Mr	Bundessmt für Wehrtechnik und Beschaffung, Luftfahrtgerät der Bundeswehr, Landshuter Allee 162a, 8000 München 19, FRG
JUANOLE, G. Dr	Laboratoire d'Automatique et d'Analyse des Systèmes du C.N.R.S., 7 Ave. du Colonel Roche, 31400 Toulouse, France
KENNIS, F. Col.	Belgian Airstaff - VDT/B, Rue d'Evere 1, 1140 Bruxelles, Belgium
KIRSTETTER, B. Dr	Eurocontrol, rue de la Loi 72, B-1040 Bruxelles, Belgium
KISTER, H. Mr	VDO-Luftfahrtgeräte Werk, Am der Sandelmühle 13, 6000 Frankfurt-Heddernheim, FRG
KLEIH, W. Dr	Messerschmitt-Bölkow-Blohm GmbH, FE 411, Postfach 801160, 8000 München 80, FRG
KOLSTAD, B. Mr	Air Material Command, P.O. Box 10, 2007 Kjeller, Norway
KUHLEN, H.P. Mr	ESG Elektronik System GmbH, Postfach 800569, Vogelweideplatz 9, 8000 München 80, FRG
LAMBRAKIS, Mrs	KETA, Delta Falirou, Palaion Faliron, Athens, Greece
LECOQ, M. Ms	MATRA SA, 37 Av L. Breguet, 78140 Velizy, France
LE GAC, J.Y. Mr	DTEN/STEN, Bureau Guidage-Pilotage, 26 Boulevard Victor, 75015 Paris, France
LIE, O. Mr	Air Material Command, P.O. Box 10, 2007 Kjeller, Norway
LIVESEY, J. Dr	School of Information & Computer Sc., Georgia Institute of Technology, Atlanta, GA 30332, USA
LOHNERT, F. Mr	TU Berlin, Institut f. Technische Informatik, Sekr. HH1, Einsteinufer 35-37, 1000 Berlin 10, FRG
MACKINTOSH, I.W. Mr	Royal Signals and Radar Establishment, St Andrews Road, Great Malvern, Worcs WR14 3PS, UK
MACPHERSON, R.W. Dr	NDHQ, CRAD 19NT, 101 Colonel By Drive, Ottawa, Onatario, KIA 0K2, Canada
MAHER, S. Lt	AFWAL/FIGLB, Wright-Patterson AFB, OH 45433, USA
MANSELL, M. Mr	British Aerospace Aircraft Group, Kingston/Brough Division, Brough, North Humberside HU15 1EQ, UK

MARTIN, J.T. Mr	Bracknell Division, Ferranti Computer Systems Ltd, Western Rd, Bracknell, Berkshire RG12 1RA, UK
MARUHN, P. Mr	Postfach 1120, Bodenseewerk Geratetechnik GmbH, D-7770 Überlingen, FRG
MAYES, D.J. Mr	Smiths Industries Ltd, Bishops Cleeve, Cheltenham, Glos., UK
McCUEN, J.W.	Sr Proj. Engineer, Systems Div., JTIDS Program Office, Hughes Aircraft Co., P.O. Box 3310, TC13, A-105, Fullerton, CA 92634, USA
McTIGUE, T.V. Mr	Dept. 312 Bldg 271B, McDonnell Aircraft Co., P.O. Box 516, St Louis, MO 63166, USA
MEGNA, V.A. Mr	F-8 DFBW Program Manager, The Charles Stark Draper Lab., Inc., MS #04, 555 Technology Square, Cambridge, Mass 02139, USA
MERAUD, M. Mr	SAGEM, rue de la Tour Billy, Argenteuil 95500, France
MOIR, I. Mr	Smiths Industries, Cheltenham Div., Bishop's Cleeve, Cheltenham, Glos. GL52 4SF, UK
MOSES, K. Mr	Flight Systems Division, Bendix Corporation, Teterboro, NJ 07608, USA
MOWAT, A.R. Mr	Ferranti Ltd, Ferry Rd, Silverknowles, Edinburgh, EH4 4AD, Scotland, UK
MOXEY, C. Mr	British Aerospace Public Ltd Co., Aircraft Group Warton Division, Warton Aerodrome, Preston, Lancs PR4 1AX, UK
NELSON, H. Mr	Naval Weapons Center, Code 3115, China Lake, CA 93555, USA
O'NIELL, Mr	Code 3114, F-18 Facility Branch, US Naval Weapons Ctr., China Lake, CA 93555, USA
PAGANO, F. Mr	Oto Melara S.p.a., v. Valdicocchi 15, 19100 La Spezia, Italy
PARTRIDGE, B.W. Mr	Marconi Radar System, West Hanningfield Rd, Great Baddow, Chelmsford, Essex CM2 8HN, UK
PENERY, M.T. Mr	EMI Electronics Ltd, R&E Div., Wells, Somerset BA5 1AA, UK
PUTZKI, R. Dr	SCS GmbH, Oehleckerling 40, 2000 Hamburg 62, FRG
QUEMARD, J.P. Mr	Electronique Marcel Dassault, 55 Quai Carnot, 92214 St. Cloud, France
RICHARDS, W.R. Mr	Smiths Industries Ltd, Bishop's Cleeve, Cheltenham, Glos., UK
ROBERTS, M. Mr	British Aerospace Aircraft Group, Kingston/Brough Division, Brough, North Humberside HU15 1EQ, UK
ROSSIGNOL, O. IA	STTI/PNI, 129 rue de la Convention, 75731 Paris Cedex 15, France
SAGE, D.S. Mr	Marconi-Avionics Ltd, Monks Way, Linford Wood, Milton Keynes, UK
SALTZER, J. Prof.	Prof. Computer Science, M.I.T., Room NE 43-505, 545 Technology Square, Cambridge, Mass. 02139, USA
SANDBRAATEN, H. Major	Air Material Command, P.O. Box 10, 2007 Kjeller, Norway
SCHLICHT, E. Mr	ESG Elektronik System GmbH, Postfach 800569, Vogelweideplatz 9, 8000 München 80, FRG
SCHOLCH, J. Mr	Industrieanlagen-Betriebsgesellschaft, IABG-Einsteinstrasse Geb. 21, 8012 Ottobrunn b. München, FRG
SCHWARTZ, R.L. Mr	Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, USA
SCOTTI DI UCCIO, G.A. Dr	Proj. Mgr., Selenia S.p.a. Avionics Systems, V. dei Castelli Romani 2, Pomezia, Italy
SERRA, E. Ing.	Elletronica San Giorgio, Via Hermada 6, 16154 Genova-Sestri, Italy
SHIN, K.G. Prof.	Electronic-Comptr & Systems Eng Dept., Rensselaer Polytechnic Inst., Troy, NY 12181, USA
SMEDSRUD, P.B. Mr	A/S Kongsberg Vaapenfabrik, Boks 25, N-3601 Kongsberg, Norway
SMESTAD, T. Mr	NDRE, Div. for Electronics, P.O. Box 25, 2007 Kjeller, Norway
SORASEN, O. Mr	NDRE, Div. for Electronics, P.O. Box 25, Kjeller, Norway
SPONHOLZ, R. Mr	Bodenseewerk Geratetechnik Abt FRN-EL, Postfach 1120, D-7770 Überlingen, FRG
STERN, A.D. Mr	Mgr., Dgtl Flt Ctrls Res., MS 86-06, Boeing Military Airplane Co., P.O. Box 3707, Seattle, Wa 98124, USA

STERNANG, A. Mr	A/S Kongsberg Vaapenfabrik, Boks 25, N-3601 Kongsberg, Norway
STOEVNE, H.H. Mr	A/S Kongsberg Vaapenfabrik, Boks 25, N-3601 Kongsberg, Norway
STRADA, J.A. Cdr	Office of Naval Research, Box 39, FPO NY 09510, USA
SVOBODOVA, L. Mrs	B.P. No.105, INRIA, Domaine de Voluceau-Rocquencourt, 78153 Le Chesnay, France
SZLACHTA, M. Mr	LITEF, Der Hellige GmbH, Lorracher Str. 18 Postfach 774, 7800 Friburg, FRG
THORSEN, J.G. Mr	A/S Kongsberg Vaapenfabrik, Boks 25, N-3601 Kongsberg, Norway
TIMMERS, H.A. Ir.	National Aerospace Laboratory NLR, Anthony Fokkerweg 2, 1559 CM Amsterdam, The Netherlands
VAGNARELLI, F. L/Col. Prof.	Aeronautica Militaire, Ufficio Delegato Nazionale all'AGARD, P. le K. Manauer, 3, 00144 Roma-Eur, Italy
VAN KEUK, G. Dr	Forschungsinstitut für Funk & Math., FGAN, 5307-Wachtberg-Werthoven, FRG
VANDECASTEELE, B. Mr	B.P. 360, 78 Quai Carnot, 92214 St. Cloud, France
VASLIN, Mr	SFIM, 13 Ave Marcel Ramolfo-Garnier, 91301-Massy, France
VOGEL, M. Dr	DFVLR, D-8031 Oberpfaffenhofen, FRG
WARD, A.O. Mr	Warton Division - Warton Aerodrome, British Aerospace - Aircraft Group, Preston PR4 1AX, UK
WARR, H. Mr	EMI Ltd, Radar House, Dawley Road, Hayes, Middlesex, UK
WEISS, M. Dr	Aerospace Corporation, P.O. Box 92957, Los Angeles, CA 90009, USA
WHITEHOUSE, H.J. Mr	Naval Ocean Systems Center, Code 5303, Catalina Blvd., San Diego, CA 92152, USA
WILCOCK, G.W. Mr	EP Department, Royal Aircraft Establishment, Farnborough, Hants GU14 6TD, UK
WOLF, J.K. Prof.	Dept. of Elec. & Comp. Eng., University of Mass., Amherst, Mass 01003, USA
WRIGHT, S.M. Mr	Systems Design Office, British Aerospace, Aircraft Group, Kingston-Brough Division, Brough, North Humberside HU 15 1EQ, UK
YOUNG, N. Dr	Ultra Electronic Controls Ltd, 136 Mansfield Rd, Western Ave., London W3, UK
ZEMPOLICH, B.A.	Dep. Tech. Admin. for Command, Ctrl & Guidance Research & Tech. Gp. NASC, Naval Air Systems Command (AIR-360B), Washington D.C. 20361, USA

REPORT DOCUMENTATION PAGE			
1. Recipient's Reference	2. Originator's Reference AGARD-CP-303	3. Further Reference ISBN 92-835-0302-3	4. Security Classification of Document UNCLASSIFIED
5. Originator	Advisory Group for Aerospace Research and Development North Atlantic Treaty Organization 7 Rue Ancelle, 92200 Neuilly sur Seine, France		
6. Title	TACTICAL AIRBORNE DISTRIBUTED COMPUTING AND NETWORKS		
7. Presented at	a Meeting of the Avionics Panel held in Røros, Norway, 22-25 June, 1981.		
8. Author(s)/Editor(s) Various			9. Date October 1981
10. Author's/Editor's Address Various			11. Pages 434
12. Distribution Statement	This document is distributed in accordance with AGARD policies and regulations, which are outlined on the Outside Back Cover of all AGARD publications.		
13. Keywords/Descriptors			
<div style="display: flex; justify-content: space-between;"> <div> <p>Computer systems hardware</p> <p>Data links</p> <p>Switching theory</p> <p>Computer programs</p> </div> <div> <p>Design criteria</p> <p>Reliability</p> <p>Avionics</p> </div> </div>			
14. Abstract			
<p>These proceedings consist of the papers and discussions presented at the Avionics Panel Meeting on "Tactical Distributed Computing and Networks" held in Røros, Norway, 22-25 June 1981. The 35 papers were divided as follows, three on state-of-the-art; five on system architecture; four on system design approaches; five on software; five on fault tolerance and reliability; six on interconnection, bussing and networking; seven on applications to avionics systems.</p>			

<p>AGARD Conference Proceedings No.303 Advisory Group for Aerospace Research and Development, NATO TACTICAL AIRBORNE DISTRIBUTED COMPUTING AND NETWORKS Published October 1981 434 Pages</p> <p>These proceedings consist of the papers and discussions presented at the Avionics Panel Meeting on "Tactical Distributed Computing and Networks" held in R��ros, Norway, 22-25 June 1981. The 35 papers were divided as follows, three on state-of-the-art; five on system architecture; four on system design approaches; five on software; five on fault tolerance and reliability; six on interconnection, bussing and networking; seven on applications to avionics systems.</p> <p>ISBN 92-835-0302-3</p>	<p>AGARD-CP-303</p> <p>Computer systems hardware Data links Switching theory Computer programs Design criteria Reliability Avionics</p>	<p>AGARD Conference Proceedings No.303 Advisory Group for Aerospace Research and Development, NATO TACTICAL AIRBORNE DISTRIBUTED COMPUTING AND NETWORKS Published October 1981 434 Pages</p> <p>These proceedings consist of the papers and discussions presented at the Avionics Panel Meeting on "Tactical Distributed Computing and Networks" held in R��ros, Norway, 22-25 June 1981. The 35 papers were divided as follows, three on state-of-the-art; five on system architecture; four on system design approaches; five on software; five on fault tolerance and reliability; six on interconnection, bussing and networking; seven on applications to avionics systems.</p> <p>ISBN 92-835-0302-3</p>	<p>AGARD-CP-303</p> <p>Computer systems hardware Data links Switching theory Computer programs Design criteria Reliability Avionics</p>
<p>AGARD Conference Proceedings No.303 Advisory Group for Aerospace Research and Development, NATO TACTICAL AIRBORNE DISTRIBUTED COMPUTING AND NETWORKS Published October 1981 434 Pages</p> <p>These proceedings consist of the papers and discussions presented at the Avionics Panel Meeting on "Tactical Distributed Computing and Networks" held in R��ros, Norway, 22-25 June 1981. The 35 papers were divided as follows, three on state-of-the-art; five on system architecture; four on system design approaches; five on software; five on fault tolerance and reliability; six on interconnection, bussing and networking; seven on applications to avionics systems.</p> <p>ISBN 92-835-0302-3</p>	<p>AGARD-CP-303</p> <p>Computer systems hardware Data links Switching theory Computer programs Design criteria Reliability Avionics</p>	<p>AGARD Conference Proceedings No.303 Advisory Group for Aerospace Research and Development, NATO TACTICAL AIRBORNE DISTRIBUTED COMPUTING AND NETWORKS Published October 1981 434 Pages</p> <p>These proceedings consist of the papers and discussions presented at the Avionics Panel Meeting on "Tactical Distributed Computing and Networks" held in R��ros, Norway, 22-25 June 1981. The 35 papers were divided as follows, three on state-of-the-art; five on system architecture; four on system design approaches; five on software; five on fault tolerance and reliability; six on interconnection, bussing and networking; seven on applications to avionics systems.</p> <p>ISBN 92-835-0302-3</p>	<p>AGARD-CP-303</p> <p>Computer systems hardware Data links Switching theory Computer programs Design criteria Reliability Avionics</p>